1993

Parallel Dynamic Mesh Generation and Domain Decomposition

Poting Wu

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

Report Number:
93-075

Parallel Dynamic Mesh Generation and Domain Decomposition

Poting Wu and Elias N. Houstis
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR-93-075
December, 1993

Parallel Dynamic Mesh Generation and Domain Decomposition

Poting Wu and Elias N. Houstis

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907, U.S.A.
e-mail: wu@cs.purdue.edu

Technical Report CSD-TR-93-075
October, 1993

Abstract

The mapping of computations to parallel machines can be realized at the various data structures associated with the computations. In the case of PDE (Partial Differential Equations) based applications, we decided to formulate this problem at the discrete data structures of the underlying computation [Chri 91]. In this paper, we formulate and implement parallel mapping algorithms on distributed memory machines, including the nCUBE II and Intel iPSC/i860. The uniqueness of our parallel mapping scheme is the fact that it integrates mesh splitting (decomposition) with mesh generation. Thus, the mesh generation and mesh-splitting preprocessors are integrated into one tool that runs parallel on the targeting machine. Furthermore, the combination of the adaptive mesh refinement and the dynamic domain decomposition becomes efficient and well defined. Several algorithmic alternatives were investigated for implementing the various parts of this preprocessor [Wu 93-1], [Wu 93-2], including suitable algorithms for mesh generation [Lohn 92], [Khan 91], [Cheng 89] and mesh decomposition [Farh 93-1], [Venk 92], [Farh 93-2], [Simon 91]. Local and global mesh refinements are also supported with mesh smoothing and side swapping. Optimal domain partitioning algorithms of the mesh data were considered [Sava 91], [Vand 93].

This work was supported by NSF grants 9123502-CDA and 9202536-CCR, AFOSR F49620-92-J-0069 and PRF grant 6902003.

CONTENTS

1 Introduction

In general, the requirement to generate finite element meshes has been an obstacle to use the finite element method. However, there are many methods available today to assist in the generation of finite element meshes. This is not to say that the generation of the element meshes is no longer a major undertaking, but the situation today is better than it has been in the past. The need to generate element meshes quickly is common to a number of computational fields, especially in an adaptive finite element process. Therefore, the mapping of element meshes generation to parallel machines becomes urgent. In this paper we formulate and implement parallel mapping algorithms on distributed memory machines including the nCUBE II and Intel iPSC/i860. The uniqueness of our parallel mapping scheme is the fact that it integrates mesh splitting (decomposition) with mesh generation. Thus, the mesh generation and mesh-splitting preprocessors are integrated into one that runs parallel on the targeting machine.

2 A Methodology for Parallel Mesh Generation and Mesh Splitting

The parallel mapping scheme we propose to study contains five major steps:

1. *Generate an initial refinable background mesh:*
   An algorithm is selected to form the initial mesh. Because a fairly fine initial background mesh can be assumed, that allows division of the background mesh into subdomains of nearly equal size with a maximum difference of one. In the fourth step, we can use the same algorithm and same code to generate mesh on subdomains in a parallel manner. These algorithm and code include the mesh generator, mesh refiner, mesh smoother, and mesh side swapper.

2. *Split the initial mesh into equal-sized subdomains:*
   Several decomposition schemes are supported so that for different shapes of geometric objects we have the opportunity to test which algorithm is most optimal. In addition, a "local optimum" scheme has been developed. This scheme makes decisions on local data to split the domain into two subdomains with minimum inter-node communication during each step of the domain decomposition.

3. *Link the subdomains to form new boundaries:*
   Before parallel mesh generation can be completed, we need to form the new boundary of the subdomains that we obtained from the domain decomposition phase. Since a multi-region and new holes may be created, extra effort (polygon locating recognition) is needed in mesh generation.

4. *Generate finer element mesh in parallel:*
   Since the introduction of the quadtree node distribution data structure, we are able to obtain the refined node distribution before generating mesh in parallel. Therefore, the communication between the processor nodes will be reduced to a minimum. Furthermore, the generated mesh will contain more global smoothness than that obtainable with another approach.

5. *Minimize the bisection width between each subdomain:*
   In practice, subdomains with a large number of interface edges may be generated, even in the case of perfect initial domain partitioning. Since the problem of optimal mesh partitioning is NP-complete, even with the restriction of fixed degree graph [Garey 76], this step implements an approximate partitioning of the mesh graph instead.

Figure 2.1 shows the methodology on which the parallel mesh and mesh decomposition preprocessor is based for 2-D regions and 4-processor machine configurations.

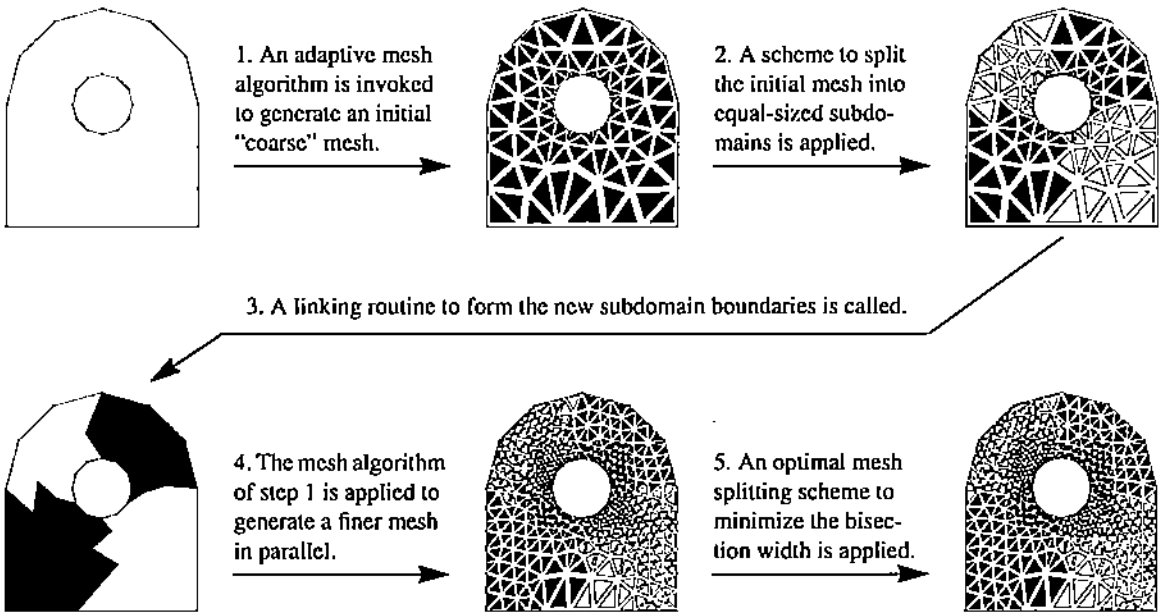


Figure 2.1: A methodology for parallel mesh and mesh-decomposition.

2.1 Integration of adaptive mesh generation and dynamic domain decomposition

The issue of partitioning element-wise versus node-wise has been discussed in numerous research articles [Venk 92]. However, because of the duality between these two approaches, we have integrated both of them as in figure 2.2. Moreover, if the solver being used is more like the node-wise decomposition, and the adaptive request is also a node-wise refinement, it is not necessary to re-generate mesh and re-decompose the subdomain. That is, it can refine the mesh only in the local region and dynamically redistribute the refined mesh to achieve load balancing. We called such a case *Homogeneity* in this figure. On the other hand, if the adaptive request is element-wise, it may need to go through the whole procedure and is called *Heterogeneity*. Since the mesh generator we developed is based on the Quadtree data structure, it is easy to refine the element mesh either by nodes or by elements. Therefore, no matter what types of solver we use, the mesh adaptation can be dynamically adjusted in both mesh generation and domain decomposition.

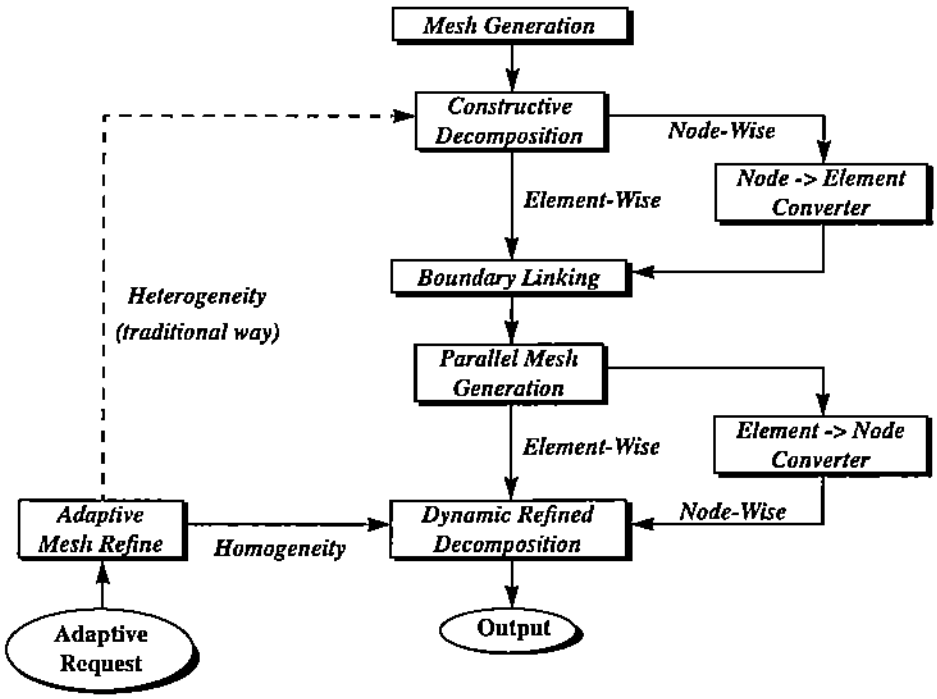


Figure 2.2: Integration of the adaptive mesh generation and the dynamic domain decomposition.

2.2 Hierarchical structure of domain decomposition

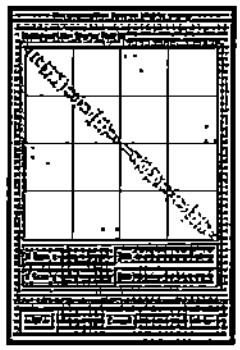
There are two major phases that request the domain decomposition in the integration. We divide the available decomposition algorithms into two categories because of their divergent features. The first type of scheme, *constructive algorithm*, constructs a partition from a given element mesh that we will describe in § 4. The other type of scheme, *refinement algorithm*, improves upon an existing partition which will be discussed in § 7. The reasons for the dividing them into two groups are:

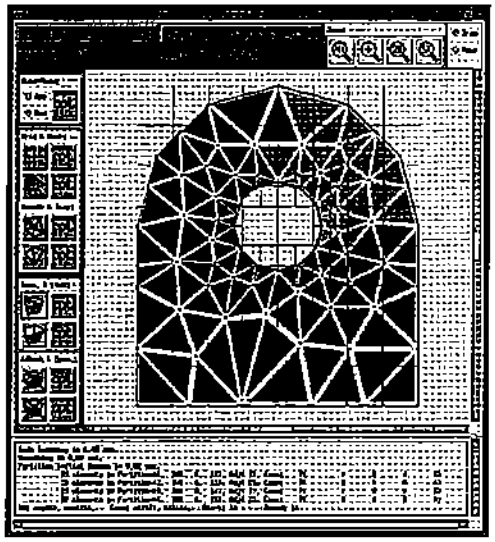
1. Based on experience with numerous experiments, applying the refinement scheme alone to an initial random partition usually gets a bad result. Moreover, search time is much longer and search direction is somehow ambiguous.

2. Since the parallel mesh generation already has an initial partition, we may choose the refinement scheme only, to reduce the re-partition time.

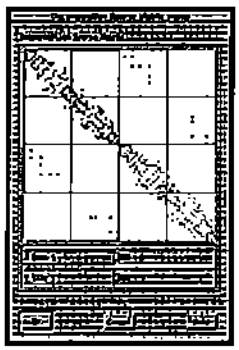
3. By hierarchically arranging various combinations of these two types of schemes we are able to examine several options and select the one with the best result.

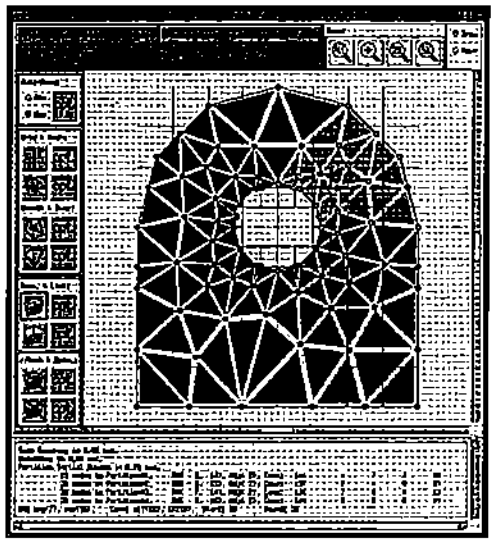
Although some algorithms are usually 'worse' than others, deciding which scheme is best depends on the target machines and problems. That is, for a specific solving environment, the 'best' characteristics such as the load balancing, interface length, and aspect ratio, are not always defining the same way. Therefore, we have integrated all the available schemes in our tool to let the users have access to the most efficient strategy for their problems.



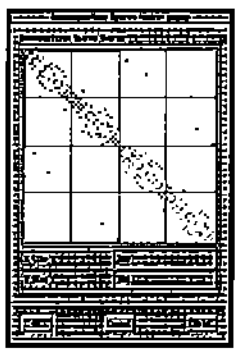


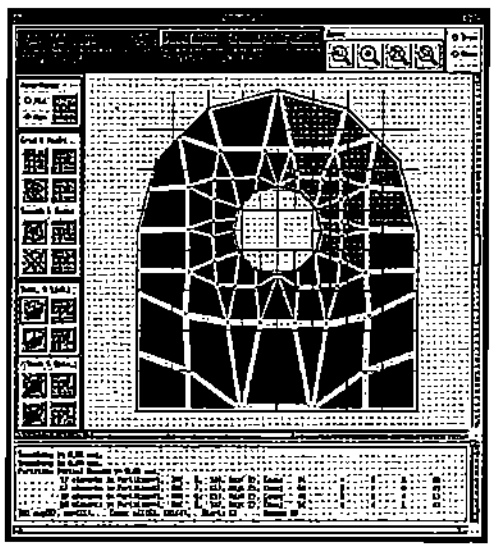
Triangular Element Mesh Generation & Element-Wise Domain Decomposition





Triangular Element Mesh Generation & Node-Wise Domain Decomposition





Quadrangular Element Mesh Generation & Element-Wise Domain Decomposition

Figure 2.3: User interface for the mesh generation and domain decomposition.

3 Formulation of the Initial Refinable Background Mesh

The methods for generation of unstructured meshes can be classified into two groups:

1. Advancing front algorithms,
2. Quadtree / Octree algorithms.

Several automatic mesh adaptation techniques of the first group can be found in the literature. The scheme described in [Khan 91] implements an adaptation scheme based on node distribution on the boundary only. The adaptation scheme introduced in [Lo 91] uses the boundary and internal contours to decide node distribution when generating element meshes. Another adaptation approach [Bykat 76], is based on the subdivision of a general polygon into convex subregions. That is, all advancing front algorithms implement mesh adaptation only on the specified or computed boundaries. They generate the internal elements of the mesh by non-adaptive or interpolating schemes.

The second family of methods are based on modifying an existing coarse mesh. The adaptation technique presented in [Cheng 89] is by means of the user specified level and vertex assignments. More discussion can be found in [Wu 93-1] and is outlined below.

For automation and generality purposes, it seems to us appropriate to pursue the use of the quadtree/octree algorithms.

1. *Automation*: Unlike the advancing front algorithms, where users need to specify the node distribution information on the object's boundary, the quadtree can automatically divide the domain into a tree structure that depends on the object's geometry. Its critical state is to maintain all the subregions simple (i.e., each subregion contains only one polygon vertex or one polygon segment).

2. *Smoothness*: Since the quadtree maintains the adjacency density to be 1/2 ratio (i.e., difference of tree level between neighbors is always no larger than one [Samet 82, 85, 89]), it manages the adaptive node distribution not only on the outer boundary of objects but also in the internal region of objects. Therefore, it provides a global smooth node distribution when generating element meshes.

3. *Adaptation*: It is normal to refine the whole domain globally or subregion locally. That is, the algorithm supports a totally controlled tree structure that decides the node distribution. Therefore, the adaptive finite element process is easy, and a user specified refined region is possible.

4. *Dynamics*: Most parallel solvers are equated on the node-distribution, while the common adaptive refinement, such as h-refine and p-refine, adjusts the element density. Hence, dynamic decomposition becomes difficult in both theory and practice. According to the well-defined quadtree data structure, "refined by node" becomes possible and natural to pursue the dynamic decomposition.

5. *Parallelism*: Because of the refining property, it has the information of global node distribution before generating element meshes in parallel. This characteristic can reduce the communication between processor nodes to a minimum.

3.1 Implementation of mesh generation scheme

The outline of the implementation of mesh generation is listed as follows. More detail can be found in [Wu 93-1].

1. *Decompose the domain into quadtree data structure*: The quadtree scheme in [Samet 84] defines the node distribution on its related hierarchical data structure. We create or modify the data structure when, i) we create the initial background quadtree and ii) we maintain a local or global refinement.

2. *Generate the element mesh*: In this stage, triangular or quadrilateral element meshes will be generated by connecting the precomputed nodes in the previous phase.

3. *Adjust the generated mesh*: One of the difficulties in unstructured FEM mesh generation is to avoid, in certain regions, degenerate elements. The usual way to solve this problem is to adjust the element and node distribution. This adjustment includes mesh smoothing and side swapping.

4. *Maintain adjacency lists*: Four types of adjacency lists are maintained in the process of mesh generation [Delj 90]. They are, i) node-node adjacency, ii) node-element adjacency, iii) element-node adjacency, and iv) element-element adjacency.

3.2 Universality of mesh element topology

The output element mesh formats in this tool include the Neutral file and Ellpack mesh. Besides, this tool supports all types of triangular and quadrilateral element topology as shown in figure 3.1.

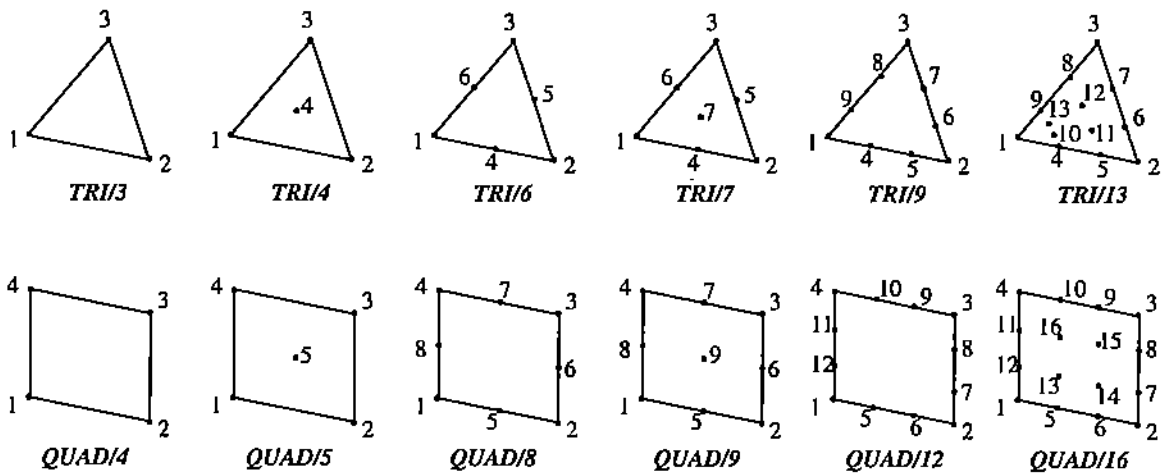


Figure 3.1: Mesh element topology.

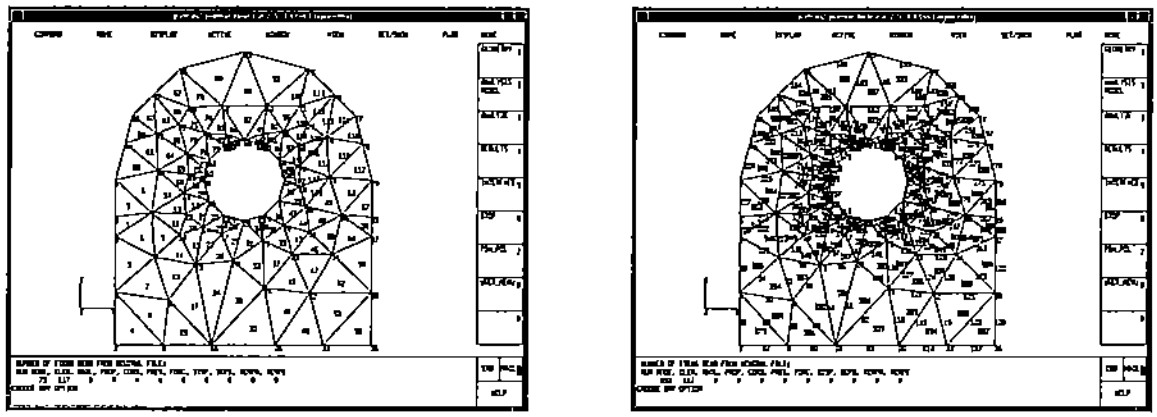


Figure 3.2: Neutral file interface for different element topology with PATRAN.

3.3 Strategy of the dynamic mesh refinement

In order to achieve the integration approach of adaptive mesh generation and dynamic domain decomposition, the available adaptive mesh refinements include:

1. *Element-wise*: One of the most commonly used element-wise strategies is the *h-refine* that includes regular division and bisection as shown in figure 3.3 [Mitch 87]:

Figure 3.3: Element-wise h-refine of a triangle by (i) regular division and (ii) bisection.

The other element-wise adjustment is the *p-refine*. Due to the availability of several element topologies and the capability of mixing different element types, it is easy to refine elements by increasing the degree of freedom.

2. *Node-wise*: We developed this type of refinement by using the well-defined quadtree data structure. Since the integration of mesh generation and domain splitting is possible, dynamic decomposition becomes efficient and easy to achieve. Furthermore, the level of computing complexity needed to make the compatible refinement is on the same order as that needed for the element-wise version. An example is shown in figure 3.4.

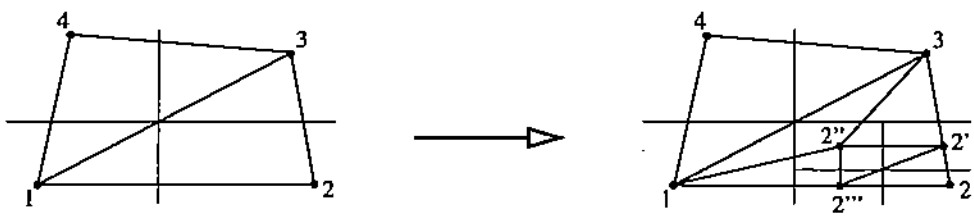


Figure 3.4: Example of node-wise h-refine.

4 Initial Constructive Domain Decomposition

For the implementation of this phase, we consider the topological graph of the initial mesh G = (V, E) where $v \in V$ represent nodes and $e \in E$ their connectivity with adjacent nodes. It appears that there are many prepared heuristics for partitioning the element mesh in balanced subdomains. In this section we present an overview of the basic ideas. The performance is reported in § 8.2.

First, we considered the class of enumerative heuristics that are based on some neighborhood search scheme utilizing the connectivity information of mesh graph G.

4.1 Algorithm 1: Neighborhood - Search schemes

For this scheme, graph G can be viewed as a tree structure. Two well known neighborhood search schemes that construct the traversal tree from graph G are the Depth-First Search (DFS) and the Breadth-First Search (BFS) [Baase 88]. The codes for DFS and BFS are included as A-4.1 and A-4.2.

In addition to the traversal order, the searching strategy maybe identified as strip-wise and domain-wise. In strip-wise decomposition, the direction in which the sorting of *edge distance* performed is fixed through the splitting procedure. In domain-wise decomposition, the splitting direction is switched according to the relative searching scheme [Farh 88]. The interpretation of the domain-wise version of the searching scheme can be found as A-4.3.

To begin the tree traverse, the starting node may be provided by the user or identified by a sequential algorithm. It is easy to realize that its selection affects the optimality of the partitioning. It can be shown that selection of the initial node is related to the *minimum bandwidth* problem. It has been shown that the maximum partitioning interface C is proportional to the maximum bandwidth. Specifically, let N_n and N_s describe the number of nodes in the graph G and number of subdomains, respectively. The following relation holds [Farh 93-1],

$$C = \frac{N_s \cdot Bandwidth}{N_n}$$

A common strategy that yields a small bandwidth or profile is the so called *pseudo-peripheral node* [Pissan 84], [Georg 79], [Gibbs 76]. The basic idea is that when you traverse the mesh tree, the maximum number of nodes per level w bounds the bandwidth of the sparse matrix from below while the upper bound is 2w - 1. Thus, to reduce the bandwidth, either we minimize the maximal width of the tree level or maximize the depth of the traversal tree. The code for a pseudo-peripheral node is included as A-4.4.

During the tree traversal procedure, the walking path in each successive level requires one to decide which child will go first, which next, and which last. One way to prevent the chance of a disconnected subdomain is to select the next node according to the increasing order of its adjacency degree [Cuth 69].

In order to avoid searching nodes in the 'long' direction of graph G, one can add extra weight to those nodes [AlNas 91]. Thus, the resulting partition will have a symmetric aspect ratio and

reduce the chance of disconnected subdomains. For the traversal tree scheme, the two normal directions can be defined as the depth and the maximal width of tree level. Hence, the extra weight can be specified as,

$$extra = \begin{cases} N_s^2 \cdot \left(\frac{level}{depth}\right) \cdot \left(\frac{depth}{width} - 1\right) & \text{if depth > width} \\ N_s^2 \cdot \left(\frac{level}{depth}\right) \cdot \left(1 - \frac{width}{depth}\right) & \text{if width > depth} \end{cases}$$

Instances of the above defined neighborhood-search scheme include the *Reverse Cuthill-McKee (RCM)* [Chan 80], [Georg 78] (it uses BFS and the pseudo-peripheral node for selecting the first node), the *Farhat Greedy* [Farh 88] (this is the domain-wise scheme with RCM), and *Al-Nasra Greedy* [AlNas 91] (this is the same as the Farhat Greedy with the added feature of maintaining the aspect ratio).

4.2 Algorithm 2: Eigenvector Spectral Search

In an eigenvector spectral search, meshes are visited in the order of increasing eigenvector value of the Laplacian matrix of the graph. Fiedler recognized that the significance of λ_2 as a measure of the connectivity of graph [Fied 73], [Fied 75-1], [Fied 75-2]. We start by defining the *Laplacian matrix* L(G) of the graph as [Barn 93],

$$L_{ij} = \begin{cases} -1 & \text{if } (v_i, v_j) \in E \\ d_i & \text{if } i = j, \text{ where } d_i \text{ is the degree of } v_i \\ 0 & \text{otherwise} \end{cases}$$

Assign a partitioning variable x_i to each node v_i such that $x_i = \pm 1$, and the balancing condition of the two subdomains will be $\Sigma x_i = 0$. After relaxing the discreteness constraint on x_i ($= \pm 1$) of the mini-cut partitioning problem and forming the approximately continuous decomposition as an optimal problem,

$$\begin{array}{ll} \text{Minimize} & 1/4\, x^T L\, x \\ \text{Subject to} & x^T \mathbf{1} = 0, \qquad x^T x = N_n . \end{array}$$

Thus, the *spectral distance* for each node can be mapped from its corresponding value of the *Fiedler vector* λ_2, where $0 = \lambda_1 < \lambda_2 \leq \lambda_3 \ldots \leq \lambda_n$ for a connected graph. The implementation of the bisection is simply finding the median node among the vector values.

The partition is formed in the strip-wise version by sorting the Fiedler vector of the graph and inducing the required number of subdomains once. The domain-wise version, the recursive bisection is carried out to a fixed 2^r pieces.

An alternate way to improve the eigenvector spectral scheme performance in run time is a multilevel implementation of typical examples introduced by Barnard [Barn 93]. This scheme requires three additional steps to be considered a single-level algorithm: Contraction, Interpolation, and Refinement.

Moreover, in addition to using the *Fiedler vector* λ_2, Hendrickson [Hend 93] also considered combining the use of the other eigenvectors λ_3, λ_4, ..., λ_n to reduce the computing cost and the communication overhead. Hendrickson's algorithm assigns the partitioning variable x_i to each v_i such that $x_i = (\pm 1, \pm 1)$ for the *spectral quadrisection* by using the vectors of λ_2 and λ_3, and $x_i = (\pm 1, \pm 1, \pm 1)$ for the *spectral octasection* by using the vectors of λ_2, λ_3, and λ_4.

Instances of the above defined eigenvector spectral search include the *Recursive Spectral Bisection (RSB)* [Simon 91] (this is the original domain-wise version of eigenvector spectral search), the *Strip-wise Eigenvector Spectral* [Venk 92] (a strip-wise version that computes the Fiedler vector once and divides the domain into required number of subdomains), the *Multilevel RSB (MRSB)* [Barn 93] (this is the multilevel implementation of the domain-wise recursive spectral bisection), and the *Recursive Spectral Quadrisection/Octasection (RSQ/RSO)* [Hend 93] (which combines the use of the eigenvectors in addition to the Fiedler vector to improve the original bisection strategy).

4.3 Domain - Axis Splitting schemes

Next, we consider another class of enumerative schemes that ignore the connectivity information of the mesh graph G. We have implemented three algorithms from this class. They are based on the domain splitting along the different types of coordinate axis or the symmetric inertia axis of mesh graph G.

4.3.1 Algorithm 3: Cartesian Axis Splitting

In this algorithm the domain splitting is along the Cartesian axis after sorting the X, Y, Z coordinates of nodes or the centre of mass of the elements. Several variations of this scheme are presented for selecting the suitable subdomains in the case of different geometry.

For different interpretation, the number of divisions in each dimension may be fixed or flexible. As we have discussed in the previous section, for the fixed division decomposition that is similar to the strip-wise scheme, the direction in which the sorting of coordinates performed is fixed through each dimensional splitting procedure. That is, for the 2-d case we have fixed numbers of rows and columns, apply one sorting on Y direction then one sorting on X direction for each row, and vice versa (see A-4.5). For the flexible one, the splitting direction is switched for each recursive step according to the relative bisection scheme.

Splitting the domain can be done by either non-recursive multi-section or recursive bisection. For the non-recursive version, the partition is formed by sorting the coordinates for each dimension and inducing the required number of subdomains. In the recursive version, the recursive bisection is carried out until the number of subdomains reaches a predefined number (see A-4.6).

Furthermore, the bisection direction may depend on matching the fixed division number, or may be simply switched for each single step. There is a local optimum approximation to select the 'best' bisection direction for each single recursive step. We can choose this direction by determining the longest expansion of domain to preserve a good aspect ratio of subdomains [Simon 91] as shown in A-4.7.

We have developed another more expensive but better performed way to select the bisection direction. It compares the 'communication cost' produced by all possible dimensions, then chooses the one that causes the least cost between these two new generated subdomains. More detail can be found in A-4.8.

4.3.2 Algorithm 4: Polar/Spherical Axis Splitting

Polar/spherical axis decomposition is similar to Cartesian axis splitting. The only difference is it splits the domain along the polar/spherical axis by sorting the R, Θ, Z/α coordinates of nodes or the centre of mass of the elements. That is, it is an approach to the *boundary-conforming curvilinear coordinate* system that is defined by axes of i) the coordinate lines conforming to the domain boundary, and ii) the other curvilinear coordinate to the angular direction. In addition to all the available options in Cartesian axis splitting, various definitions of the original point are possible. This algorithm can be described as follows [Lori 88]:

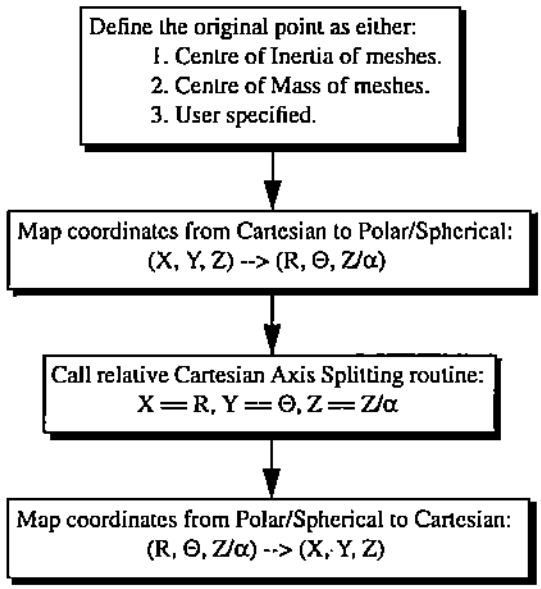


Figure 4.1: Strategy for polar/spherical axis splitting.

However, since the range of angle is within (-π, π] and there is a jump from π to -π, there is a high probability that the partition this scheme generates may be disconnected. To overcome the disadvantage, we shift the disconnected part by adding 2π as follows:

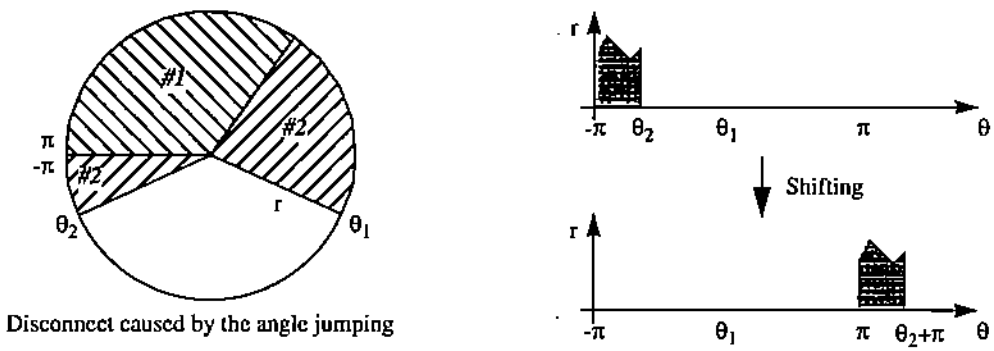


Figure 4.2: Avoid jumping disconnected subdomain by angle shifting.

Moreover, when selecting the bisection direction by the longest expansion, this scheme also needs to convert the dimensionless angle into length by locating θ_1 and θ_2 for computing its angle expansion, then multiplying by $2r_{avg}$ to obtain the comparable length.

When mapping one coordinate system to another coordinate system, we may define the original point in the following three possible ways. They are: i) Centre of Inertia of nodes or the coordinate of the elements, ii) Centre of Mass of nodes or the coordinate of the elements, or iii) User specified.

4.3.3 Algorithm 5: Inertia Axis Splitting

This scheme first pre-computes the main symmetry axis according to the coordinate of nodes or the centre of mass of elements [Lori 88]. Then, it splits the domain into several subdomains along the axis. It repeats this step until a predefined number of subdomains is reached. One common way to compute the main symmetry axis is to compute the eigenvector corresponding to the largest eigenvalue of *inertia matrix* $I = A^TA$ [Willia 92],

$$I = A^TA = \begin{bmatrix} \sum x_i^2 & \sum x_i y_i & \sum x_i z_i \\ \sum y_i x_i & \sum y_i^2 & \sum y_i z_i \\ \sum z_i x_i & \sum z_i y_i & \sum z_i^2 \end{bmatrix}$$

where $A_{Nn\times 3}$ is the matrix of the mesh coordinates. The algorithm in A-4.9 shows the splitting scheme of the inertia axis decomposition.

Again, for this decomposition scheme we have both the strip-wise and domain-wise options. In the strip-wise version, the partition is formed by sorting the angle along the main symmetry axis and dividing into the required number of subdomains. In the domain-wise version, the recursive bisection is repeated until the number of subdomains has reached a fixed predefined number.

Furthermore, we may define the main symmetry axis as usually but not always the eigenvector corresponding to the largest eigenvalue (that is, the axis of minimum moment of inertia). Alternately, we could choose any of the principal inertia directions of the graph. That is, they are the three eigenvectors of either A^TA or the 3×3 inertia matrix that is defined in [Farh 93-1]:

$$I = \begin{bmatrix} \sum\left(y_i^2 + z_i^2\right) & -\sum x_i y_i & -\sum x_i z_i \\ -\sum y_i x_i & \sum\left(z_i^2 + x_i^2\right) & -\sum y_i z_i \\ -\sum z_i x_i & -\sum z_i y_i & \sum\left(x_i^2 + y_i^2\right) \end{bmatrix}$$

Actually, eigenvectors and eigenvalues of these two definitions are related. For the 2-d case the first eigenvalue of the first definition is equal to the second eigenvalue of the second definition, and vice versa.

For this scheme, when computing the inertia matrix, there are various ways to define the coordinate of the original point. The three available specifications are: i) Centre of Inertia of mesh in current domain, ii) Centre of Mass of mesh in current domain, or iii) User specified.

4.4 Algorithm 6: Scattered Mapping schemes

This partitioning strategy may or may not use the connectivity information, depending on the mapping scheme used. The basic feature of a scattered scheme is to construct the partitioning suitable for mapping to a hypercube computer so that the *communication distance* is minimal.

The scattered scheme maps a regular, rectangular mesh onto a rectangular processor lattice to make a load balancing and minimal bisection width partitioning that also has minimal communication distance. According to the definition, a single copy of the fundamental processor lattice is called a *template*, and a processor's assignment within a single template is called a *patch*. A scattered subdomain will be assigned many disconnected patches throughout the domain, and each patch is in a unique template [Mori 87]. Therefore, Each subdomain boundary in the mesh contains only the nearest neighbor communication as shown in figure 4.3:

12 1100	4 0100	6 0110	14 1110	12 1100	4 0100	6 0110	14 1110
8 1000	0 0000	2 0010	10 1010	8 1000	0 0000	2 0010	10 1010
9 1001	1 0001	3 0011	11 1011	9 1001	1 0001	3 0011	11 1011
13 1101	5 0101	7 0111	15 1111	13 1101	5 0101	7 0111	15 1111
12 1100	4 0100	6 0110	14 1110	12 1100	4 0100	6 0110	14 1110
8 1000	0 0000	2 0010	10 1010	8 1000	0 0000	2 0010	10 1010
9 1001	1 0001	3 0011	11 1011	9 1001	1 0001	3 0011	11 1011
13 1101	5 0101	7 0111	15 1111	13 1101	5 0101	7 0111	15 1111

Figure 4.3: Scattered subdomains of 2×2 templates & 4×4 patches.

As we can see, a hypercube computer of dimension d is a 2^d processor machine with each processor connecting to other d processors, and they differ in precisely one bit position. Moreover, the mapping above is always a wrap around communication channel, in the fashion of a torus. That is, each of the left-most processors can communicate to the right-most at a distance of one, and also the upper-most one to the lower-most one. Therefore, all communication distances under this arrangement are exactly one. The mapping of processor lattice is easy to compute by the *Gray code* that is formed by a Hamiltonian cycle of the d-bit binary numbers [Leigh 92].

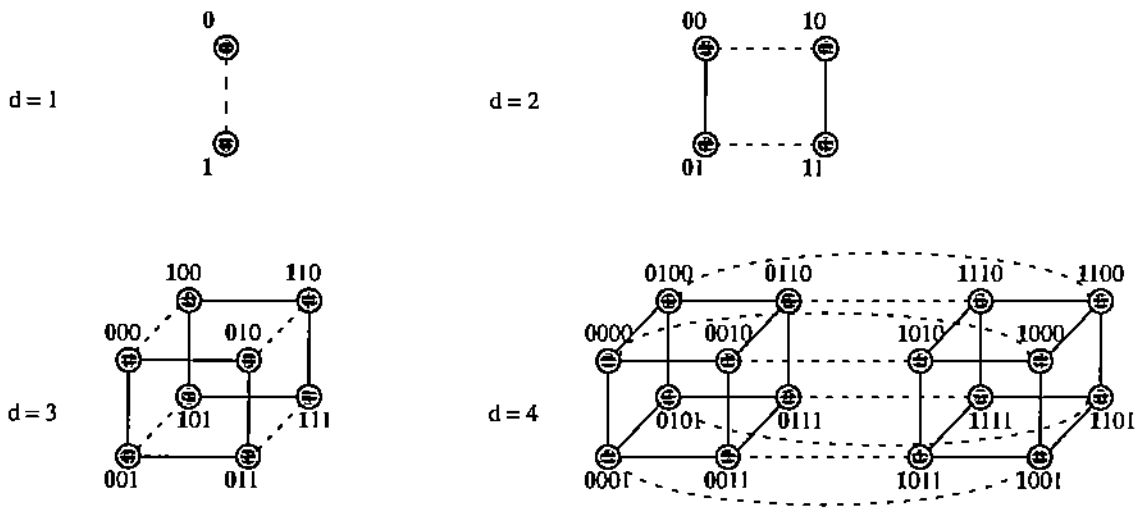


Figure 4.4: The d-node hypercube for d = 1, 2, 3, and 4. Dimension 1 edges are shown in dashed.

However, the issue of mapping irregular mesh onto a regular, rectangular mesh that may or may not use the connectivity information is beyond the scope of this paper.

4.5 Extended improvement of mesh splitting algorithm

In addition to the variant options on these decomposition algorithms, there is still more room to improve the splitting result. We have made the following three extended improvements on the existing algorithms.

1. *Local and global refinement*: For the algorithms that construct the partition in the bisecting fashion, it is possible to improve the bisection slightly by considering local modification among the set of cutting edges. The available refinement schemes will be described in § 7. After the final partition is constructed, a suitable multi-section refinement scheme can be selected to smooth the boundary between each subdomain and reduce the inter-node communication or unbalancing load.

2. *Weighted load balancing*: A more flexible approach for load balancing is to consider the computing weight of each node. Most decomposition algorithms operate under the assumption that all of the nodes have the same work loading on parallel distributing computation. Therefore, load balancing is meant to construct the mesh into equal-sized subdomains with a maximum size difference of one. However, the work associated with each node in the parallel solver is usually not the same. For instance, the computing load for each node in the sparse matrix-vector multiplication is proportional to the degree of freedom of the node [Leete 93]. Hence, a more general approach for the weighted load balancing scheme is as follows:

$$minimize \left| \sum_{v_i \in V_1} w_i - \sum_{v_j \in V_2} w_j \right|$$

The definition of the weight associated with each node could be:

(a) $w_i = 1$ for each node. This is the special case of the un-weighted version.

(b) w_i = DOF (number of adjacent nodes) for each node.

Actually, node-wise partition with w_i = DOF has the same load balancing effect as element-wise partition with $w_i = 1$, and vice versa.

3. *Arbitrary number of subdomains*: The restriction of the subdomains number N_s is a power of two and has no problem on a hypercube computer. However, most recent and proposed parallel machines are grid-based and the need for breaking the domain into an arbitrary number is required then. We have implemented this flexibility by modifying the previous weighting requirement when bisecting N_s domain into N_{s1} and N_{s2} as follows:

$$minimize \left| \frac{\sum_{v_i \in V_1} w_i}{N_{s_1}} - \frac{\sum_{v_j \in V_2} w_j}{N_{s_2}} \right| \qquad \begin{array}{l} , N_s \text{ is even: } N_{s1} = N_{s2} = 1/2\, N_s \\ \text{odd: } |\, N_{s1} - N_{s2}\, | = 1 \end{array}$$

When splitting N_n nodes into N_s subdomains, we compute the number of nodes N_{nk} in sub-domain k by the following formula in order to force the maximum difference of size to remain equal to one:

$$N_{n_k} = \frac{k \cdot N_n}{N_s} - \sum_{i=1}^{k-1} N_{n_i} \qquad , k = 1, \ldots, N_s$$

5 Subdomain Boundary Linking

After mesh decomposition, we need a linking routine to connect the new boundary for each subdomain before proceeding with the final mesh generation in parallel. In some cases domain splitting can generate more than one subdomain in a single processor. Also, new holes may be created. Therefore, the linking routine needs to separate the outer boundary polygon from the hole polygon and identify the hole polygon that belongs to the outer boundary polygon. The linking algorithm is as follows (more detail can be found in A-5.1 and A-5.2):

1. Locate all the polygon lists including outer boundary polygon and hole polygon. The searching strategy needs the element-element adjacency list to identify the boundary element, and the node-element adjacency list to locate the next boundary element.

2. Determinate the outer boundary polygon and the hole polygon by the polygon locating recognition algorithm [Wu 93-1].

The natural way to form the new boundary of subdomains is to partition the domain in element-wise before linking. Since our decomposition tool has both node-wise and element-wise capabilities, the linking routine needs a preprocessor to connect the node-wise partitioning. We can either i) make a brand new linking process that constructs the boundary by connecting the mid-point of the interface edges, or ii) convert the element-wise partitioning to node-wise partitioning before the normal linking routine. For the latter scheme, the preprocessor can determine the coloring of each element by its adjacent nodes efficiently for the conversion, then smooth the boundary by applying the refinement decomposition algorithms for the local optimum.

6 Final Parallel Mesh Generation

For the parallel mesh generation various strategies have been proposed. One strategy [Lohn 92] is to generate the mesh of each subdomain in parallel and then generate the mesh of the inter-subdomain region sequentially. Alternately, one can generate the mesh of the inter-subdomain region sequentially and then generate the mesh of each subdomain in parallel. In both cases, these strategies need to communicate between processor nodes for generating mesh in the inter-subdomain region, or generating them sequentially. Furthermore, the generated mesh in each subdomain does not always have a global smooth node distribution.

In our proposed approach, we have introduced the quadtree data structure to supervise the node distribution. Thus, it is easy and efficient to refine the quadtree globally before generating the mesh in parallel. Therefore, during the generation of the parallel mesh, there is no need for communication between processors. Furthermore, the global smoothness of the node distribution assures more uniform mesh elements.

In addition, we can use the same algorithm and a sequential mesh generator to generate the unstructured mesh on each subdomain in parallel. Similarly, the parallel local mesh smoothing and side swapping can be done by the use of sequential global mesh smoothing and side swapping. Thus, our approach accommodates the resemblance of the parallel codes to existing sequential mesh generation codes.

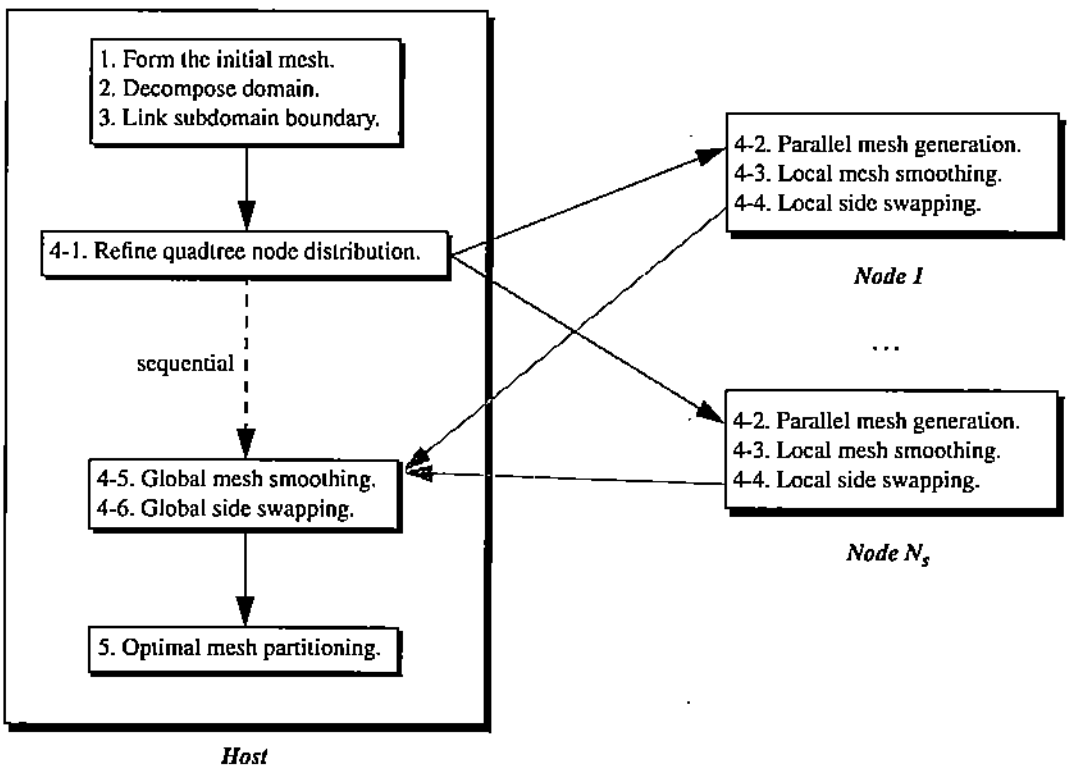


Figure 6.1: A strategy for the parallel final mesh generation.

7 Final Refined Domain Decomposition

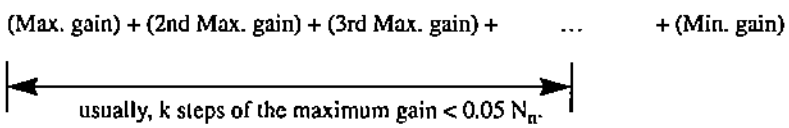
In practice, parallel mesh generators can generate local meshes with a large number of edges joining the sets between subdomains. These cases can appear even when trying to generate perfectly initial domain splittings. In this phase, we have considered, so far, two types of partitioning algorithms [Krish 84]. The first type is the one considered in § 4 and is called *constructive algorithm* since the partitioning of the domain is based on a given finite element mesh. The second type is one which improves upon an existing partitioning, called *refinement algorithm* that we will discuss in this section.

In this step domain decomposition schemes are needed to approximate the minimum bisection width [Sava 91], [Vand 93]. Four widely used strategies for this problem are the *Kernighan-Lin (KL) algorithm* [Krish 84], [Kern 70], the *Simulated Annealing (SA) algorithm* [John 89], [Kirk 83], the *Stochastic Evolution (SE) algorithm* [Saab 91], and the *Tabu Search (TS) algorithm* [Hertz 87], [Glov 85].

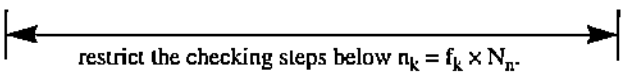
7.1 Algorithm 1: Kernighan - Lin Heuristic algorithm

The Kernighan-Lin scheme (see A-7.1) attempts to locate the best k exchange pairs of nodes by constructing a sequence of gains. This searching procedure may continue even though some local gains are negative. Therefore, it can locally identify the swapping sequence that will produce the maximum gain. Since it searches all pairs of nodes to make the improvement as large as possible, it also increases much of the executing time [Kern 70]. We have developed a methodology that notably improves the search time.

According to the original KL algorithm, to search the best set of exchange pairs, we need to check the pair with the maximum gain, then the pair with the second maximum gain, and finally the pair with the minimum gain. Then choose step number k to maximize the accumulated gain as,



From many of the experiments, the optimal step ratio $r_k = k / N_n$ is usually less than 0.05. That is, 95% of the work is redundant. Therefore, to prevent the waste, we introduce a K-L factor f_k to restrict the searching region under $n_k = f_k \times N_n$ steps as,



Consequently, from this scheme of restricted searching steps, we usually get the same result and a *speedup* of about 300.

Moreover, because of the property of the connected graph, the swapping nodes of the Kernighan-Lin are usually the interface nodes between two subdomains. Hence, for a good initial already-grouping partitioning, checking the boundary node is sufficient to get the same result. Therefore, this approach needs only about 1/n of the executing time.

After parallel mesh generation, the number of element meshes generated by the parallel scheme in each subdomain may be different. One way to reduce the cost of unbalancing is to move the extra nodes from the larger set to the smaller set before the Kernighan-Lin scheme. We select the moving nodes with the maximum cost, and only the nodes on the boundary, to prevent the creating of disconnected graph. Since the cost function is used in both the balancing preprocess and the Kernighan-Lin scheme, this combination can easily reduce the computing and developing cost in a natural way.

For the partitioning optimization of multiple subdomains, we have the multi-domain version (see A-7.2) that repeatedly selects pairs of subsets until no more swapping is needed.

The Kernighan-Lin scheme makes small local improvement only by *downhill moves* until no such alternation yields a better solution to reach a local optimal partitioning. In order to avoid a poor locally optimal partitioning, the simulated annealing, the stochastic evolution, and the Tabu search algorithms occasionally allow the *uphill moves* to randomize this procedure. Therefore, the search algorithms could prevent the refinement stuck in a globally poor mesh partitioning.

7.2 Algorithm 2: Simulated Annealing algorithm

The Simulated Annealing (SA) algorithm was introduced by Kirkpatrick [Kirk 83]. It is an approach that attempts to prevent a poor local optimum by allowing an occasional uphill move. The basic idea is done under a random number generator and a *temperature* control parameter. That is, a new partitioning is accepted either when the gain is larger than or equal to zero (a downhill move), or with probability $e^{gain/T}$ (an uphill move). As the temperature drops to zero according to the cooling schedule, the probability of allowing an uphill move will also become smaller. Therefore, the SA scheme has large uphill probability at the beginning, when the optimal point is related far away, and decreases the chance of uphill movement at the final steps, when it is near the optimum. The related pseudo-code is listed as A-7.3.

Since the original SA scheme makes the convergence very slow, it is difficult to solve the optimum problem in practice. However, there is a double loop revised version as shown in A-7.4 that usually converges more quickly compared to the primary one [Saab 91], [John 89].

7.3 Algorithm 3: Stochastic Evolution algorithm

An *evolution concept* has been suggested by Saab [Saab 91] to solve the partitioning optimum problem. That is, the Stochastic Evolution (SE) algorithm is an *adaptive heuristic* scheme that allows the uphill move probability to be updated whenever it is necessary. For instance, if the current gain is not larger than zero, the scheme will increase the uphill move probability. On the other hand, if the scheme got a better partitioning, it rewards itself by increasing the iteration number. Under this assumption, the algorithm expects to yield good solutions at a fast executing speed. For more detail of this scheme, the pseudo-code can be found in A-7.5.

7.4 Algorithm 4: Tabu Search algorithm

The unique feature of the Tabu Searching (TS) is the construction of a Tabu list. It is a list of Tabu moves that prevents the searching procedure from being cyclically stuck; these are moves that are not permitted at the current iteration because that would bring the searching back to where it was at some previous state. The searching iteration will update the Tabu list by inserting the newest move into the end of the cyclical list T and removing the oldest move from T. The algorithm shown in A-7.6 may accept the uphill moves only if they have not occurred before, to prevent an endless cycle. In [Hertz 87], [Golv 85], the suggested size |T| of the Tabu list is 7. Smaller values may still cause cycling and larger values do not make the uphill moves effective.

Other Stochastic techniques, such as neural networks, have being considered in [Byun 93] and we plan to apply these for solving the above decomposition problem.

7.5 Algorithm 5: Parallel Mob Heuristic algorithm

Since these heuristics schemes we have described are hard to parallelize, Savage [Sava 91] introduced a parallel heuristic scheme called the Mob heuristic (MOB). It has many of the features of the previously discussed algorithms but is better able to exploit available parallelism. Instead of searching neighbors with one swapping only, Mob deterministically swaps large numbers of nodes between two partitionings. It selects the large number of 'good' random nodes on each subdomain in parallel for further swapping as shown in A-7.7.

7.6 Formulation of dynamic domain decomposition

The issue of the dynamic domain decomposition has been discussed in [Willa 91], [Walsh 93], [Curra 92]. This approach is relatively important for time-dependent problems where frequent remeshing may occur and the local mesh adjustment is quite small. According to the integration of our adaptive mesh generation and the dynamic domain decomposition, both the developing and solving environments are well-defined in concept and practice. A dynamic repartitioning approach that was introduced in [Walsh 93] is clustering the internal nodes in each subdomain as a single node, and the associated weight is counted with the multiplicity of the number of nodes contained in it. Since it greatly reduces the total number of nodes in the decomposition procedure, the execution time is much less than the static repartitioning.

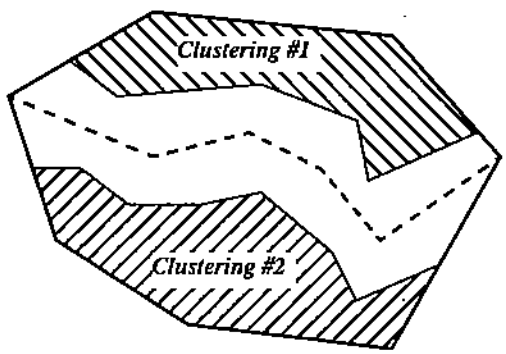


Figure 7.1: Clustering approach for the dynamic domain decomposition.

However, the scheme may fail to make the load balancing dynamically because sometimes the bisection could fall in the middle of a cluster in the sorting process. Moreover, an expensive iterative technique is needed to decide the terminate status.

One method we developed to avoid the medium node falling in the middle of a cluster is using the refinement algorithm we just discussed in this section, instead of using the sorting based constructive schemes in § 4. Therefore, not only is the failure prevented, but also the result is much smoother.

The other strategy we proposed to apply for dynamic load balancing is based on the concept that for all constructive decomposition algorithms, there is virtually a list of sorting order. For instance, the X/Y/Z coordinate ordering in axis-domain splitting, or the searching distance ordering in neighborhood search schemes. The basic idea is that we can dynamically 'shift' the boundary separators in such a sorting list to maintain load balancing. Moreover, one can preserve the validity of this ordering list by inserting elements when the mesh has reached refinement, or removing elements when the mesh has coarsened.

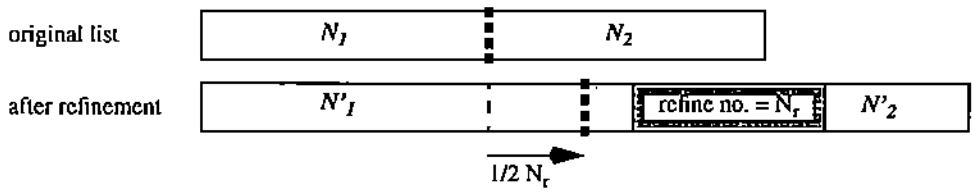


Figure 7.2: Order-shifting approach for the dynamic domain decomposition.

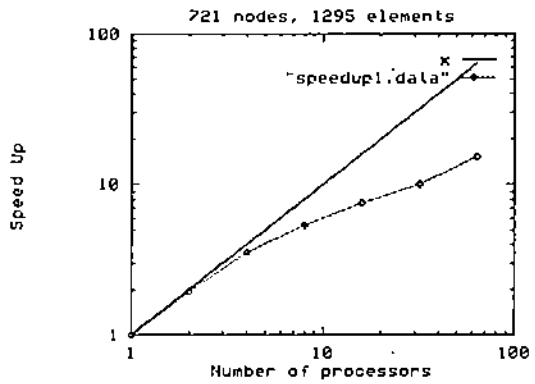
8 Performance of Parallel Mesh Generation and Domain Decomposition

8.1 Performance of parallel mesh generation

For the two examples, engine rod head and torque arm, we give some preliminary performance data including *Speedup* and *Utilization* that shows the three states - busy, overhead, and idle - as a function of time for each processor. We categorize each processor as *idle* if it has suspended execution awaiting a message that has not yet arrived or if it has ceased execution at the end of the run, *overhead* if it is executing the communication stuff in program, and *busy* if it is executing a portion of the program other than the communication stuff [Geist 92], [Geist 90], [Heat 93-1], [Heat 93-2], [Heat 91].

8.1.1 Example 1 -- Engine rod head

Figure 8.1 and 8.3 shows not only the *speedup* but also their *utilization count* and *utilization summary* by a graphical display system, ParaGraph, for visualizing the behavior and performance of parallel mesh generation on message-passing architecture.



Utilization Count

states of idle, overhead, and busy as function of time.

Utilization Summary

overall cumulative percentage of time in idle, overhead, and busy states.

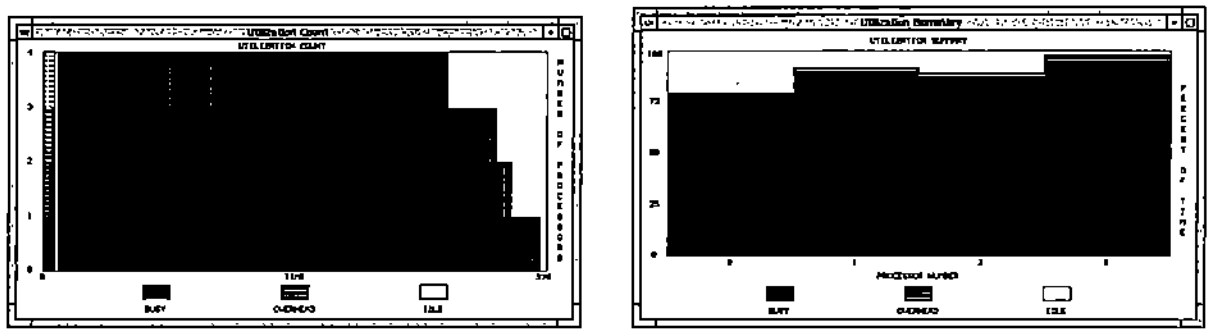


Figure 8.1: Performance of parallel mesh generation - Engine rod head.

8.1.2 Example 2 -- Torque arm

Figure 8.2 depicts a torque arm that we have used to evaluate the computational behavior of our parallel mesh generator.

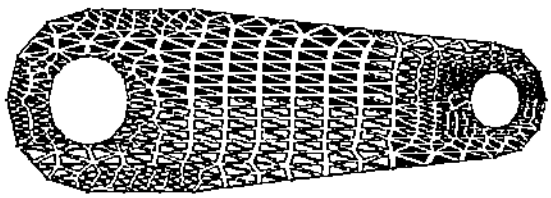
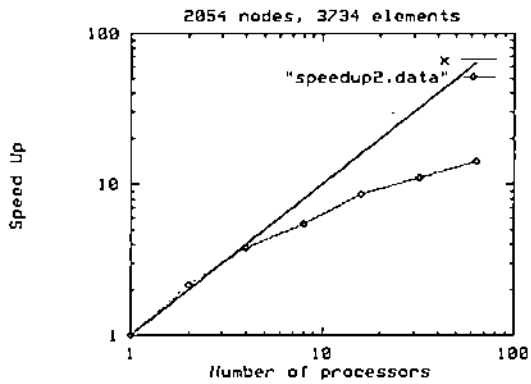


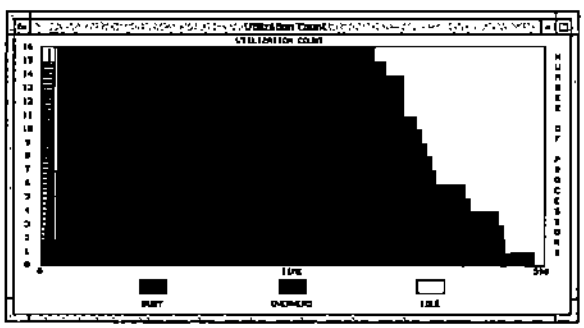
Figure 8.2: Finite element mesh for the torque arm.

Figure 8.3 presents various performance indicators of the parallel mesh generation for the torque arm.



Utilization Count

states of idle, overhead, and busy as function of time.



Utilization Summary

overall cumulative percentage of time in idle, overhead, and busy states.

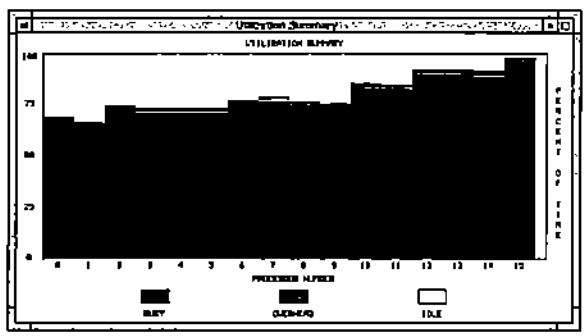


Figure 8.3: Performance of parallel mesh generation - Torque arm.

8.2 Performance of constructive mesh splitting algorithm

The goal of an "optimal" mesh splitting algorithm is to partition a mesh of elements into several equal-sized sets (*Load Balancing*) so that the number of nodes joining the sets is the minimum (*Minimax Interface Length*). In addition to these two measurements, the algorithm needs to keep low synchronization costs by reducing the number of adjacent partitions for each subdomain (*Minimax Adjacent Partitions Number*). It has been shown [Chri 91] that such partitioning leads to optimal mapping of the underlying computations.

Load Balancing:

There is the same number of mesh elements in each subdomain. Therefore, each processor node takes the same solving time in a parallel manner. We measure the performance of each splitting algorithm by computing:

$$\max_{i,j} \left(|N_i - N_j| \right) / N_{total} \qquad i = 1, \ldots, p \quad j = 1, \ldots, p \quad i \neq j$$

Minimax Interface Length:

Minimize the maximum interface length so that there is as little communication as possible between processor nodes. We measure the performance of each splitting algorithm by computing:

$$max \sum_{i=1}^{P} C_{i,j} \qquad j = 1, \ldots, p \quad i \neq j$$

$$\max C_{i,j} \qquad i = 1, \ldots, p \quad j = 1, \ldots, p \quad i \neq j$$

where $C_{i,j}$ denotes the number of common edges or nodes between subdomains i and j.

Another measurement of the interface communication cost can be defined by the *interpartition boundary vertices (IBV)* [Venk 92], since inter-processor communication is taken between subdomains only through the IBVs that are shared by different processors.

Minimax Adjacent Partitions Number:

The number of adjacent partitions for each subdomain is related to the start-up costs. We can consider this number as a weight to the interface communication cost as:

$$max\left(N_{b,j} \cdot \sum_{i=1}^{P} C_{i,j} \right) \qquad j = 1, \ldots, p \quad i \neq j$$

Or simply add these two costs by a predefined factor β;

$$\beta \cdot max\,(N_{b,j}) + max \sum_{i=1}^{P} C_{i,j} \qquad j = 1, \ldots, p \quad i \neq j$$

where $N_{b,i}$ denotes the number of adjacent partitions number of subdomain i.

Tables 8.1 and 8.2 indicate the performance, communication cost and start-up cost, of the considered mesh splitting algorithms for seven different meshes.

Applications			EC	EC	ER	ER	ER	ER	TA	Reference
(mesh size , machine configuration)			(724m,4p)	(724m,16p)	(117m,4p)	(531m,16p)	(2191m,4p)	(2191m,16p)	(3734m,16p)	
Neighborhood - Search	DFS	basic	69 / 39	46 / 14	17 / 10	31 / 12	222 / 114	127 / 59	175 / 96	
		degree ordering	99 / 41	46 / 16	16 / 11	27 / 11	118 / 71	104 / 40	141 / 68	
	BFS	strip-wise	59 / 34	77 / 42	15 / 8	51 / 27	94 / 53	111 / 58	133 / 67	RCM
		domain-wise	33 / 12	25 / 9	8 / 4	25 / 10	56 / 35	53 / 19	81 / 23	Greedy
		aspect ratio	33 / 12	27 / 10	10 / 5	26 / 9	56 / 35	60 / 16	64 / 24	Al-Nasra 91
Eigenvector Spectral			24 / 13	19 / 8	7 / 5	17 / 7	43 / 23	36 / 14	53 / 21	MRSB
Domain - Axis Splitting	Cartesian	recur. bisection	19 / 10	26 / 9	7 / 4	21 / 9	37 / 21	48 / 19	53 / 20	
		longest expan.	21 / 11	25 / 9	7 / 4	21 / 9	37 / 21	48 / 19	50 / 27	
		local optimum	21 / 11	25 / 9	7 / 4	24 / 10	37 / 21	49 / 20	53 / 18	Wu 93
	Polar	recur. bisection	43 / 28	39 / 16	16 / 11	29 / 14	70 / 38	63 / 27	65 / 30	Loriot 88
		longest expan.	19 / 10	25 / 13	8 / 5	21 / 11	38 / 25	46 / 23	73 / 27	
		local optimum	19 / 10	20 / 8	8 / 5	17 / 7	38 / 25	35 / 17	61 / 23	Wu 93
	Inertia	first eigenvector	50 / 28	57 / 30	12 / 6	32 / 13	57 / 39	67 / 30	126 / 54	
		last eigenvector	45 / 23	34 / 16	15 / 8	31 / 15	77 / 45	65 / 30	112 / 61	Loriot 88

Table 8.1: This table lists the pairs p = ($max \sum_{i=1}^{P} C_{i,j}$ / max $C_{i,j}$) of various splitting algorithms for different (mesh size, machine configuration) pairs and three applications. (EC = Engine cap, ER = Engine rod head, TA = Torque arm.)

Applications			EC	EC	ER	ER	ER	ER	TA	Reference
(mesh size , machine configuration)			(724m,4p)	(724m,16p)	(117m,4p)	(531m,16p)	(2191m,4p)	(2191m,16p)	(3734m,16p)	
Neighborhood - Search	DFS	basic	207 / 112	322 / 222	51 / 28	162 / 175	666 / 251	762 / 540	2100 / 1035	
		degree ordering	297 / 145	308 / 230	48 / 27	140 / 148	354 / 192	810 / 504	1060 / 753	
	BFS	strip-wise	118 / 81	154 / 363	30 / 22	102 / 326	188 / 139	222 / 710	266 / 683	RCM
		domain-wise	99 / 53	150 / 151	16 / 15	175 / 146	112 / 101	371 / 305	729 / 341	Greedy
		aspect ratio	99 / 53	162 / 153	30 / 17	208 / 149	112 / 101	413 / 309	495 / 345	Al-Nasra 91
Eigenvector Spectral			48 / 47	95 / 125	14 / 12	85 / 105	86 / 77	180 / 223	265 / 248	MRSB
Domain - Axis Splitting	Cartesian	recur. bisection	38 / 38	115 / 150	14 / 14	105 / 129	74 / 71	240 / 301	318 / 284	
		longest expan.	42 / 42	120 / 152	14 / 14	105 / 129	74 / 71	240 / 301	200 / 276	
		local optimum	42 / 42	120 / 152	14 / 14	96 / 130	74 / 71	220 / 270	318 / 254	Wu 93
	Polar	recur. bisection	129 / 74	273 / 188	48 / 29	182 / 159	210 / 128	413 / 340	288 / 297	Loriot 88
		longest expan.	38 / 38	50 / 159	16 / 15	63 / 127	76 / 69	138 / 270	365 / 308	
		local optimum	38 / 38	80 / 124	16 / 15	70 / 109	76 / 69	165 / 236	305 / 254	Wu 93
	Inertia	first eigenvector	100 / 74	255 / 287	36 / 19	192 / 160	171 / 93	402 / 329	756 / 597	
		last eigenvector	90 / 63	170 / 151	45 / 21	155 / 167	231 / 93	325 / 345	390 / 481	Loriot 88

Table 8.2: This table lists the pairs p = ($max\left(N_{b,j} \cdot \sum_{i=1}^{P} C_{i,j} \right)$ / interpartition boundary vertices (IBV)) of various splitting algorithms for different (mesh size, machine configuration) pairs and three applications. (EC = Engine cap, ER = Engine rod head, TA = Torque arm.)

Since the domain-axis decomposition does not make use of the connectivity information, the partition it creates may be disconnected. Consequently, it increases the interface length greatly, and the disconnected partition may also have the undesirable effect of increasing the number of its neighbor subdomains. The basic BFS and DFS techniques produce long interface partitions because the searching schemes define the interface level as a set of concentric circles. As expected, if the search starts from the center of graph, the interface length grows at the rate of searching distance, or at the rate of the square root of the partitioning number as shown below,

$$C_{i,i+1} \cong \sqrt{\frac{i}{j}} \cdot C_{j,j+1}$$

However, it can be improved by the domain-wise approach, as does the Greedy search. Theoretically, the eigenvector spectral algorithm produces the partition most uniform and connected with the smallest interface length. In light of the time consuming eigenvector calculation of the Lanczos method, the desire for parallel implementation has become urgent. The approaches include the parallel version of the Lanczos method [Leete 93] or the divide-and-conquer parallelism on the bisection of the graph.

The domain-wise option that is available in most algorithms usually produces partitions that have more neighbor subdomains but smaller interface length, while the strip-wise produces long, thin partitions that have fewer neighbor subdomains but larger interface length. One can determine which scheme is more suitable by weighing the interface communication cost and the start-up cost.

Although the overall performance of interface communication is usually not very good in the minimax bandwidth approach, the result of partition can always produce the 'best' sparse matrix on the distributing computation, theoretically. That is because this scheme does not only construct a 'block-wise optimum' but also an 'element-wise optimum' as shown in figure 8.4. Hence, in spite of the fact that the interface length may larger than that of the other approach, the sparse matrix related to the minimax bandwidth scheme is usually considered better for computing time and memory storage. Besides the schemes in this section for finding the minimax bandwidth, many approaches (such as by deterministic or random exchanging rows and columns of sparse matrix) can be found in [Rosen 68], [Alway 65], [Akyuz 68]. However, according to their characteristics, they can be related to the refinement algorithms we described in § 7.

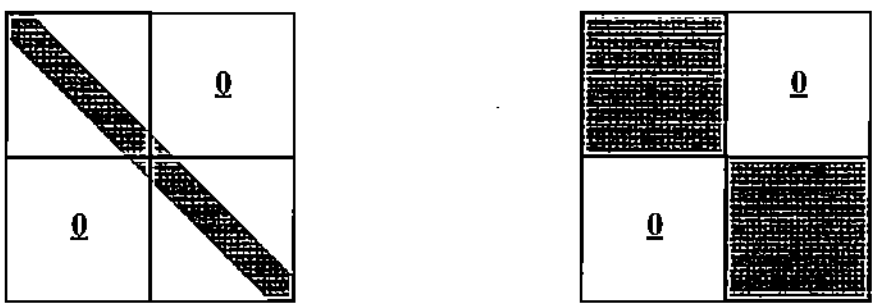


Figure 8.4: Sparse matrices related to the minimax bandwidth algorithm and the others.

Consequently, the measurements for the performance of various decomposition algorithms could include not only load balancing, communication, and start-up cost, but also the bandwidth of the sparse matrix on the distributing computation. We have developed a user interface for evaluating the corresponding sparse matrix in our integration. In addition to the numerical values of the maximal and average length of bandwidth in each subdomain, the pictorial interface of the sparse matrix can clearly illustrate all the splitting performance in the following ways:

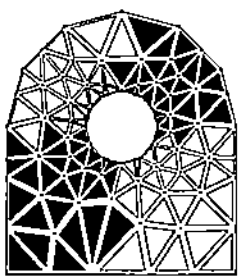
1. *Load Balancing*: Since the size of each subdomain is proportional to the number of elements in each sub-matrix of the sparse matrix, the more uniform sized distribution of the sub-block, the better load balancing performance is.

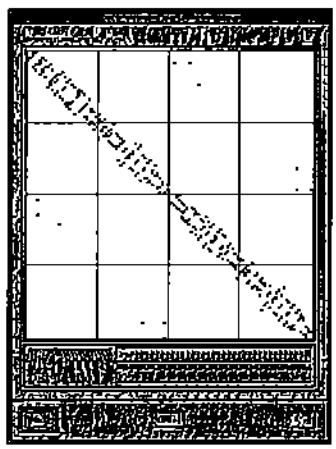
2. *Communication Length*: For each subdomain, communication length is related to the number of elements which fall into those non-diagonal submatrics in its corresponding row or column.

3. *Interpartition Boundary Vertices (IBV)*: The total number of IBV is the number of elements that fall into the non-diagonal submatrices.

4. *Adjacent Subdomain*: The number of adjacent partitions for each subdomain is the number of non-diagonal submatrices containing elements inside its corresponding row or column.

5. *Bandwidth of Sparse Matrix*: The bandwidth of sparse matrix for each subdomain on the distributing computation can be evaluated by the element distribution in its diagonal submatrix.





Load Balancing: 29 / 29 / 29 / 30
Communication: 7 / 4 Bandwidth: 6 / 15
Start-Up Cost: 14 IBV: 14

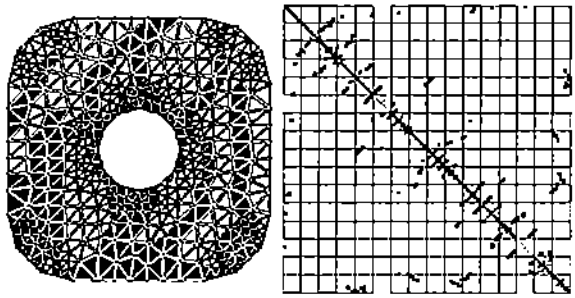
Figure 8.5: User interface and performance evaluation of parse matrices.

Surprisingly, the BFS scheme we expected to have the best bandwidth performance is only good on domains without any hole and is heavily dependent on the starting search location. Although the eigenvector spectral scheme usually produces partitions with small communication length, the corresponding bandwidth performance is almost the worst. Neither the fixed nor the variable bandwidth sparse matrix schemes can apply on the scheme. Fortunately, the local-optimum domain-axis splitting algorithm we developed not only has the same degree of communication and start-up performance as the eigenvector spectral scheme, but also has good bandwidth performance, even better than the BFS scheme.

Tables 8.3 indicates the performance, length of bandwidth, of the considered mesh splitting algorithms for seven different meshes

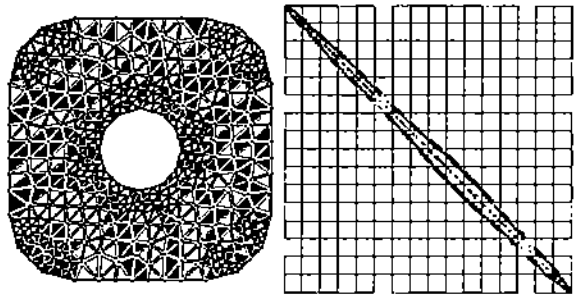
Applications			EC	EC	ER	ER	ER	ER	TA	Reference
(mesh size , machine configuration)			(724m,4p)	(724m,16p)	(117m,4p)	(531m,16p)	(2191m,4p)	(2191m,16p)	(3734m,16p)	
Neighborhood - Search	DFS	basic	18 / 164	5 / 41	4 / 24	3 / 20	46 / 419	14 / 111	12 / 212	
		degree ordering	13 / 106	5 / 42	3 / 23	5 / 30	45 / 446	13 / 109	15 / 212	
	BFS	strip-wise	33 / 65	19 / 45	9 / 16	14 / 31	65 / 92	50 / 91	53 / 120	RCM
		domain-wise	21 / 41	9 / 24	7 / 15	7 / 19	39 / 77	18 / 48	23 / 58	Greedy
		aspect ratio	21 / 42	9 / 25	7 / 16	8 / 20	39 / 77	18 / 48	24 / 67	Al-Nasra 91
Eigenvector Spectral			25 / 136	11 / 44	7 / 22	7 / 33	44 / 529	19 / 136	29 / 229	MRSB
Domain - Axis Splitting	Cartesian	recur. bisection	16 / 34	7 / 17	6 / 17	6 / 17	30 / 102	15 / 35	23 / 103	
		longest expan.	16 / 34	7 / 17	6 / 17	6 / 17	30 / 102	15 / 35	24 / 122	
		local optimum	16 / 34	7 / 17	6 / 15	8 / 29	30 / 102	18 / 101	25 / 88	Wu 93
	Polar	recur. bisection	8 / 22	3 / 9	2 / 6	3 / 14	15 / 56	7 / 25	18 / 101	Loriot 88
		longest expan.	16 / 50	12 / 43	5 / 11	9 / 27	31 / 83	21 / 75	20 / 100	
		local optimum	16 / 50	8 / 22	5 / 11	7 / 18	31 / 83	16 / 50	23 / 120	Wu 93
	Inertia	first eigenvector	32 / 93	13 / 42	7 / 17	9 / 32	43 / 141	22 / 86	41 / 190	
		last eigenvector	25 / 86	10 / 33	7 / 22	9 / 30	46 / 174	25 / 106	42 / 203	Loriot 88

Table 8.3: This table lists the pairs p = (Average Bandwidth / Maximum Bandwidth) in local subdomain of various splitting algorithms for different (mesh size, machine configuration) pairs and three applications. (EC = Engine cap, ER = Engine rod head, TA = Torque arm.)



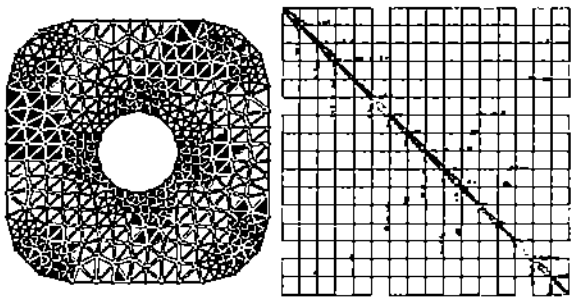
DFS - basic

Communication: 46 / 14 Bandwidth: 5 / 41
Start-Up Cost: 322 IBV: 222



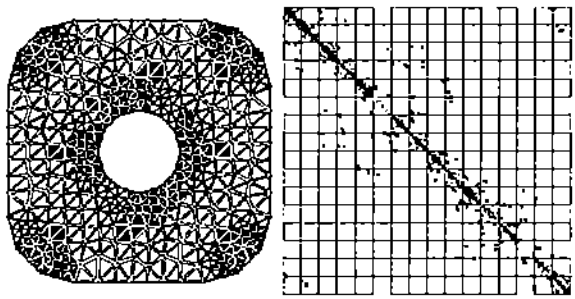
BFS - strip-wise

Communication: 77 / 42 Bandwidth: 19 / 45
Start-Up Cost: 154 IBV: 363



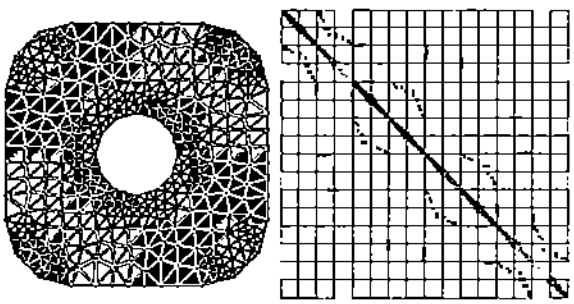
BFS - domain-wise

Communication: 25 / 9 Bandwidth: 9 / 24
Start-Up Cost: 150 IBV: 151



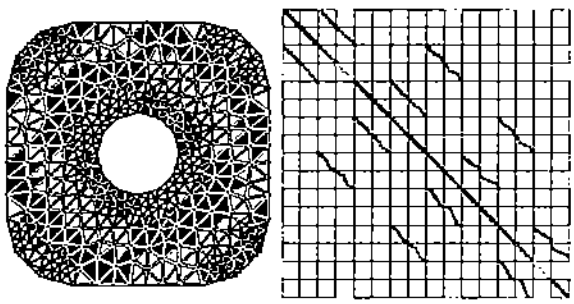
Eigenvector Spectral

Communication: 19 / 8 Bandwidth: 11 / 44
Start-Up Cost: 95 IBV: 125



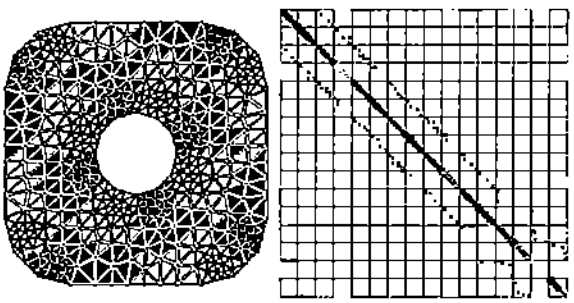
Cartesian - local optimum

Communication: 25 / 9 Bandwidth: 7 / 17
Start-Up Cost: 120 IBV: 152



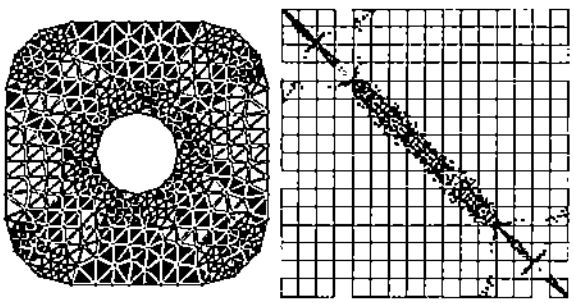
Polar - recursive bisection

Communication: 39 / 16 Bandwidth: 3 / 9
Start-Up Cost: 273 IBV: 188



Polar - local optimum

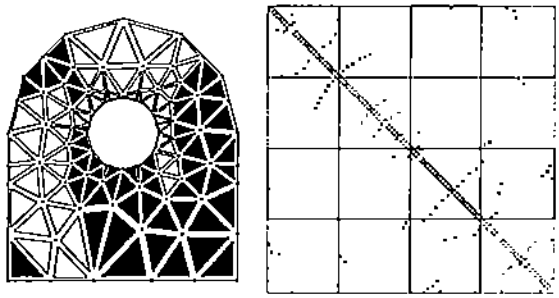
Communication: 20 / 8 Bandwidth: 8 / 22
Start-Up Cost: 80 IBV: 124



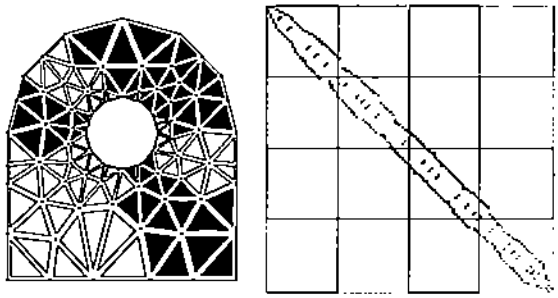
Inertia - first eigenvector

Communication: 57 / 30 Bandwidth: 13 / 42
Start-Up Cost: 255 IBV: 287

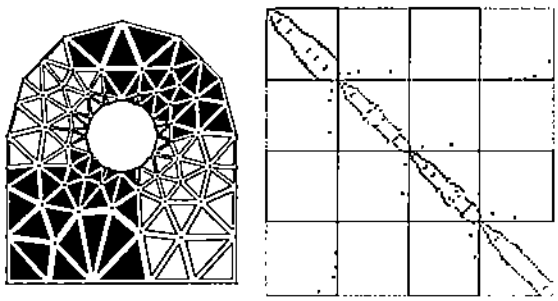
Figure 8.6: The performance and sparse matrix of the considered mesh splitting algorithms for the engine cap (724m, 16p).



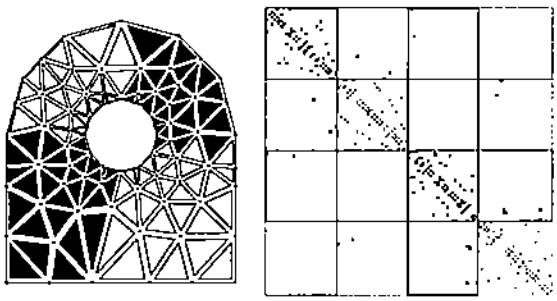
DFS - basic
Communication: 17 / 10 Bandwidth: 4 / 24
Start-Up Cost: 51 IBV: 28



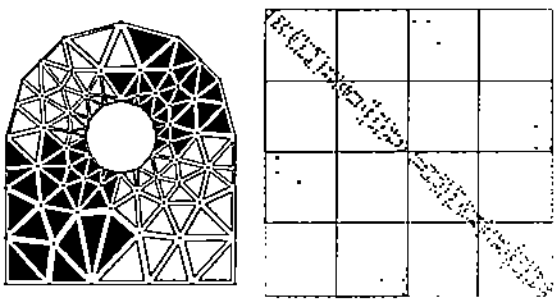
BFS - strip-wise
Communication: 15 / 8 Bandwidth: 9 / 16
Start-Up Cost: 30 IBV: 22



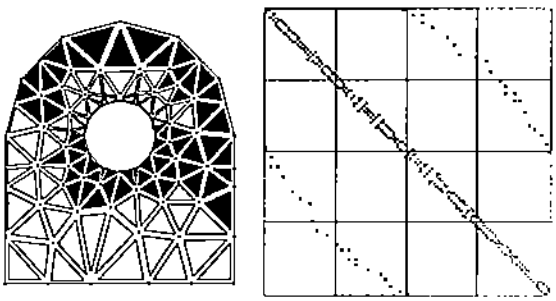
BFS - domain-wise
Communication: 8 / 4 Bandwidth: 7 / 15
Start-Up Cost: 16 IBV: 15



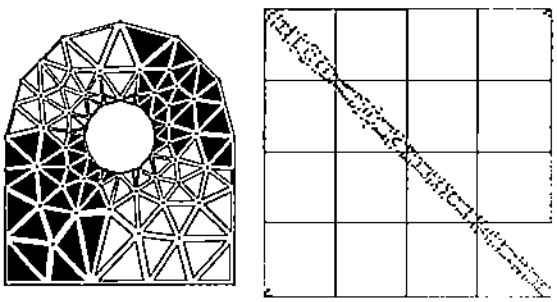
Eigenvector Spectral
Communication: 7 / 5 Bandwidth: 7 / 22
Start-Up Cost: 14 IBV: 12



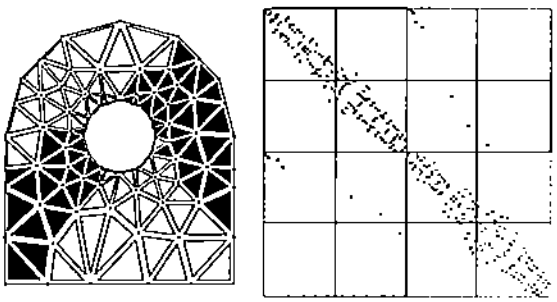
Cartesian - local optimum
Communication: 7 / 4 Bandwidth: 6 / 15
Start-Up Cost: 14 IBV: 14



Polar - recursive bisection
Communication: 16 / 11 Bandwidth: 2 / 6
Start-Up Cost: 48 IBV: 29



Polar - local optimum
Communication: 8 / 5 Bandwidth: 5 / 11
Start-Up Cost: 16 IBV: 15



Inertia - first eigenvector
Communication: 12 / 6 Bandwidth: 7 / 17
Start-Up Cost: 36 IBV: 19

Figure 8.7: The performance and sparse matrix of the considered mesh splitting algorithms for the engine rod head (117m, 4p).

8.3 Performance of refined mesh splitting algorithm

Tables 8.4, 8.5 and 8.6 indicate the improvement by the Kernighan-Lin refinement compared with tables 8.1, 8.2 and 8.3.

Applications			EC	EC	ER	ER	ER	ER	TA	Reference
(mesh size , machine configuration)			(724m,4p)	(724m,16p)	(117m,4p)	(531m,16p)	(2191m,4p)	(2191m,16p)	(3734m,16p)	
Neighborhood - Search	DFS	basic	57 / 29	46 / 12	15 / 8	27 / 10	132 / 46	119 / 59	122 / 59	
		degree ordering	70 / 32	46 / 16	15 / 11	27 / 11	142 / 57	82 / 39	132 / 60	
	BFS	strip-wise	40 / 24	33 / 13	13 / 7	24 / 8	74 / 41	82 / 42	100 / 50	RCM
		domain-wise	24 / 8	20 / 7	6 / 3	23 / 8	48 / 27	40 / 17	65 / 18	Greedy
		aspect ratio	24 / 8	21 / 7	7 / 3	24 / 7	48 / 29	39 / 13	45 / 17	Al-Nasra 91
Eigenvector Spectral			20 / 11	18 / 7	5 / 3	16 / 6	37 / 21	34 / 13	41 / 16	MRSB
Domain - Axis Splitting	Cartesian	recur. bisection	14 / 7	21 / 7	5 / 3	16 / 6	26 / 14	38 / 13	47 / 16	
		longest expan.	13 / 7	17 / 7	5 / 3	16 / 6	26 / 14	38 / 13	37 / 21	
		local optimum	13 / 7	19 / 7	5 / 3	17 / 6	27 / 14	38 / 15	36 / 18	Wu 93
	Polar	recur. bisection	31 / 20	30 / 13	14 / 9	25 / 9	60 / 33	55 / 23	55 / 28	Loriot 88
		longest expan.	13 / 7	18 / 9	6 / 4	22 / 10	27 / 16	42 / 19	55 / 20	
		local optimum	13 / 7	16 / 6	6 / 4	15 / 6	27 / 16	28 / 15	48 / 18	Wu 93
	Inertia	first eigenvector	42 / 22	33 / 10	10 / 5	23 / 10	45 / 30	46 / 25	92 / 39	
		last eigenvector	33 / 17	23 / 10	14 / 8	26 / 11	70 / 41	58 / 26	82 / 43	Loriot 88

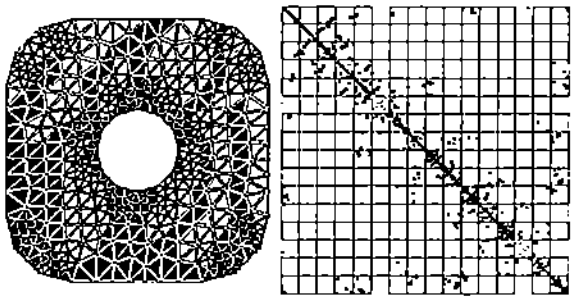
Table 8.4: This table lists the pairs p = ($max \sum_{i=1}^{P} C_{i,j}$ / max $C_{i,j}$) of various splitting algorithms for different (mesh size, machine configuration) pairs and three applications . (EC = Engine cap, ER = Engine rod head, TA = Torque arm.)

Applications			EC	EC	ER	ER	ER	ER	TA	Reference
(mesh size , machine configuration)			(724m,4p)	(724m,16p)	(117m,4p)	(531m,16p)	(2191m,4p)	(2191m,16p)	(3734m,16p)	
Neighborhood - Search	DFS	basic	171 / 91	322 / 207	45 / 23	189 / 162	396 / 163	616 / 470	1586 / 771	
		degree ordering	210 / 99	280 / 211	45 / 25	147 / 140	426 / 172	511 / 406	1020 / 671	
	BFS	strip-wise	80 / 56	198 / 189	26 / 18	110 / 151	148 / 106	243 / 514	200 / 487	RCM
		domain-wise	72 / 39	120 / 119	12 / 12	184 / 124	96 / 80	320 / 238	585 / 262	Greedy
		aspect ratio	72 / 39	126 / 126	21 / 13	216 / 129	132 / 81	280 / 263	450 / 257	Al-Nasra 91
Eigenvector Spectral			40 / 40	90 / 113	10 / 10	80 / 96	74 / 68	170 / 195	165 / 207	MRSB
Domain - Axis Splitting	Cartesian	recur. bisection	28 / 27	105 / 115	10 / 10	64 / 105	52 / 52	190 / 223	282 / 234	
		longest expan.	26 / 25	85 / 108	10 / 10	64 / 105	52 / 52	190 / 223	148 / 221	
		local optimum	26 / 25	85 / 109	10 / 10	102 / 104	54 / 52	190 / 219	180 / 208	Wu 93
	Polar	recur. bisection	93 / 53	210 / 158	42 / 25	200 / 138	180 / 103	330 / 281	252 / 249	Loriot 88
		longest expan.	26 / 26	54 / 120	12 / 11	66 / 115	54 / 52	126 / 226	275 / 255	
		local optimum	26 / 26	80 / 100	12 / 11	70 / 97	54 / 52	135 / 195	336 / 208	Wu 93
	Inertia	first eigenvector	84 / 63	231 / 171	30 / 15	92 / 121	135 / 72	270 / 265	736 / 439	
		last eigenvector	66 / 48	115 / 119	42 / 18	125 / 141	210 / 82	290 / 292	325 / 378	Loriot 88

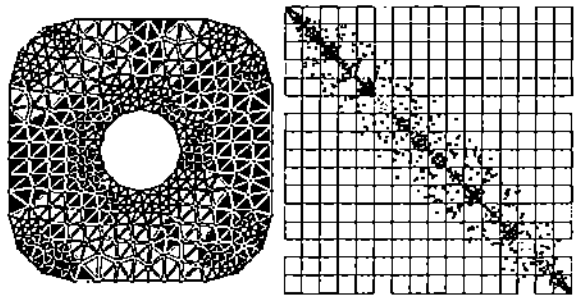
Table 8.5: This table lists the pairs p = ($max\left(N_{b,j} \cdot \sum_{i=1}^{P} C_{i,j} \right)$ / interpartition boundary vertices (IBV)) of various splitting algorithms for different (mesh size, machine configuration) pairs and three applications. (EC = Engine cap, ER = Engine rod head, TA = Torque arm.)

Applications			EC	EC	ER	ER	ER	ER	TA	Reference
(mesh size , machine configuration)			(724m,4p)	(724m,16p)	(117m,4p)	(531m,16p)	(2191m,4p)	(2191m,16p)	(3734m,16p)	
Neighborhood - Search	DFS	basic	39 / 176	12 / 42	6 / 25	8 / 31	79 / 528	28 / 135	48 / 232	
		degree ordering	45 / 173	11 / 43	6 / 29	8 / 31	48 / 499	25 / 134	35 / 231	
	BFS	strip-wise	32 / 103	16 / 44	9 / 25	12 / 32	65 / 173	45 / 133	52 / 218	RCM
		domain-wise	26 / 164	13 / 44	9 / 25	10 / 31	48 / 504	25 / 135	32 / 228	Greedy
		aspect ratio	26 / 169	12 / 44	9 / 25	10 / 32	47 / 517	27 / 133	30 / 227	Al-Nasra 91
Eigenvector Spectral			28 / 145	12 / 45	9 / 28	10 / 32	58 / 512	25 / 136	35 / 232	MRSB
Domain - Axis Splitting	Cartesian	recur. bisection	19 / 178	11 / 44	8 / 25	9 / 30	32 / 372	18 / 135	25 / 228	
		longest expan.	19 / 172	10 / 44	8 / 25	9 / 30	32 / 372	18 / 135	28 / 227	
		local optimum	19 / 172	10 / 44	8 / 28	8 / 31	33 / 546	20 / 136	29 / 233	Wu 93
	Polar	recur. bisection	18 / 179	7 / 44	6 / 26	7 / 32	28 / 545	13 / 135	24 / 231	Loriot 88
		longest expan.	21 / 170	13 / 44	6 / 24	10 / 32	43 / 536	24 / 126	24 / 220	
		local optimum	21 / 170	9 / 41	6 / 24	9 / 32	43 / 536	20 / 135	27 / 228	Wu 93
	Inertia	first eigenvector	34 / 176	12 / 44	8 / 25	9 / 30	45 / 545	23 / 130	46 / 231	
		last eigenvector	28 / 174	11 / 43	7 / 18	9 / 32	48 / 542	27 / 136	43 / 230	Loriot 88

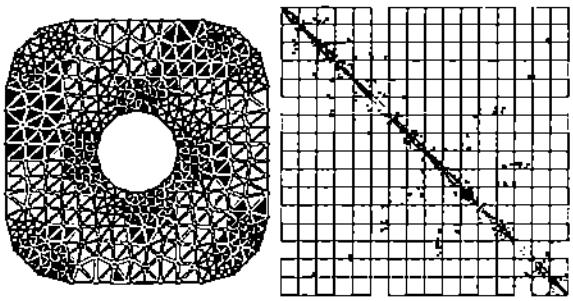
Table 8.6: This table lists the pairs p = (Average Bandwidth / Maximum Bandwidth) in local subdomain of various splitting algorithms for different (mesh size, machine configuration) pairs and three applications. (EC = Engine cap, ER = Engine rod head, TA = Torque arm.)



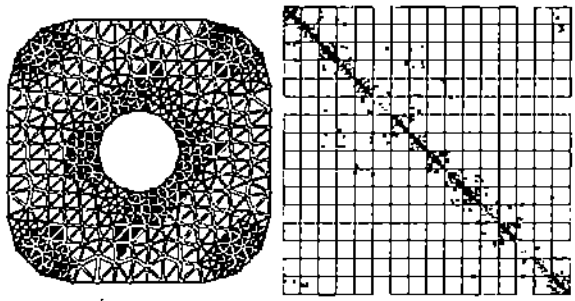
DFS - basic (KL)
Communication: 46 / 12 Bandwidth: 12 / 42
Start-Up Cost: 322 IBV: 207



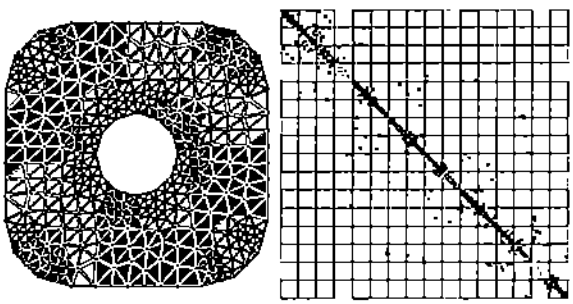
BFS - strip-wise (KL)
Communication: 33 / 13 Bandwidth: 16 / 44
Start-Up Cost: 198 IBV: 189



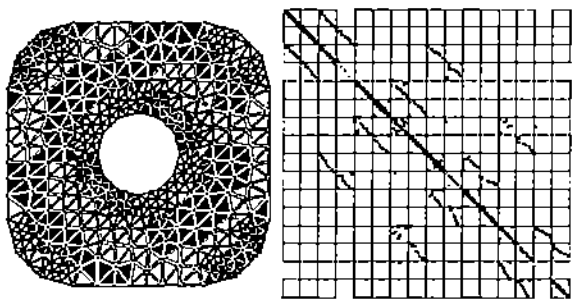
BFS - domain-wise (KL)
Communication: 20 / 7 Bandwidth: 13 / 44
Start-Up Cost: 120 IBV: 119



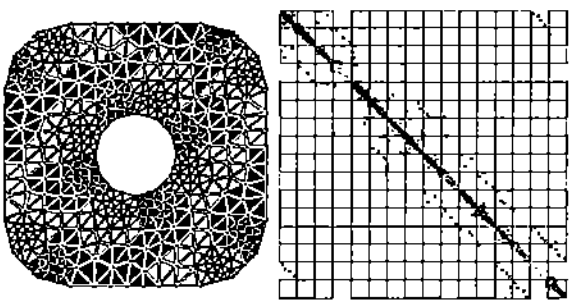
Eigenvector Spectral (KL)
Communication: 18 / 7 Bandwidth: 12 / 45
Start-Up Cost: 90 IBV: 113



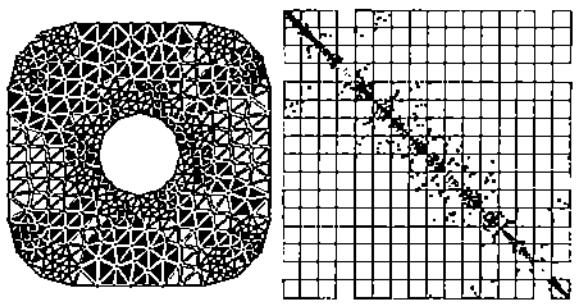
Cartesian - local optimum (KL)
Communication: 19 / 7 Bandwidth: 10 / 44
Start-Up Cost: 85 IBV: 109



Polar - recursive bisection (KL)
Communication: 30 / 13 Bandwidth: 7 / 44
Start-Up Cost: 210 IBV: 158

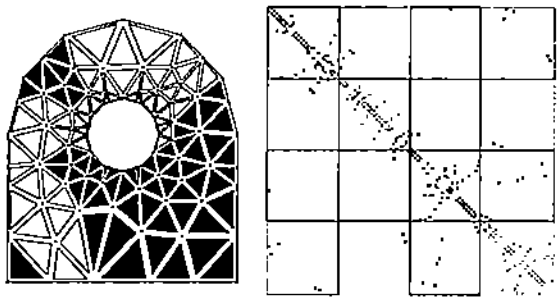


Polar - local optimum (KL)
Communication: 16 / 6 Bandwidth: 9 / 41
Start-Up Cost: 80 IBV: 100

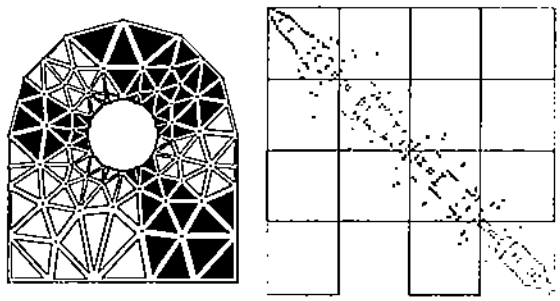


Inertia - first eigenvector (KL)
Communication: 33 / 10 Bandwidth: 12 / 44
Start-Up Cost: 231 IBV: 171

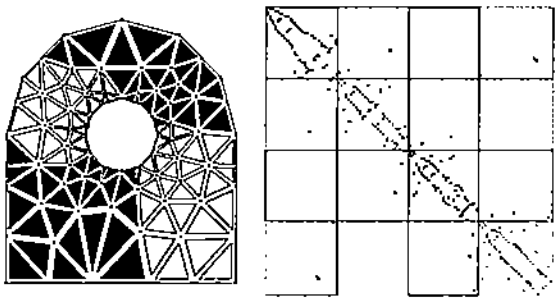
Figure 8.8: The performance and sparse matrix of the considered mesh splitting algorithms with Kernighan-Lin refinement for the engine cap (724m, 16p).



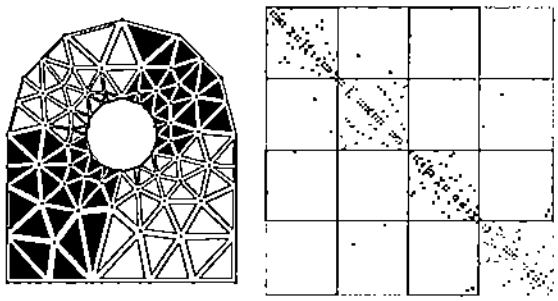
DFS - basic (KL)
Communication: 15 / 8 Bandwidth: 6 / 25
Start-Up Cost: 45 IBV: 23



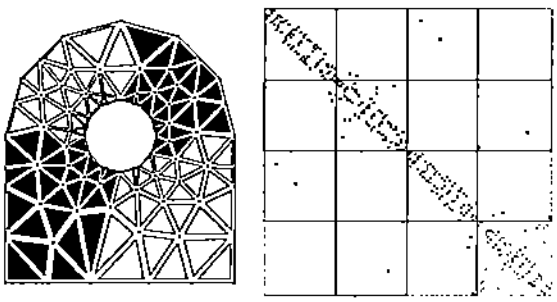
BFS - strip-wise (KL)
Communication: 13 / 7 Bandwidth: 9 / 25
Start-Up Cost: 26 IBV: 18



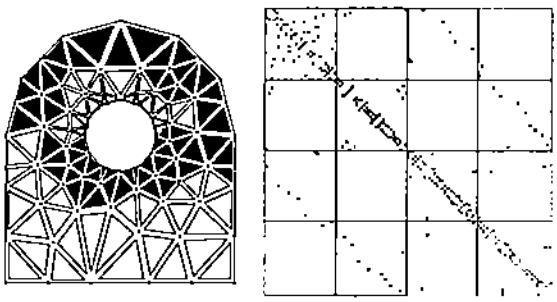
BFS - domain-wise (KL)
Communication: 6 / 3 Bandwidth: 9 / 25
Start-Up Cost: 12 IBV: 12



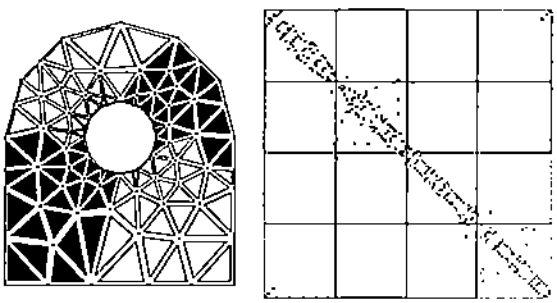
Eigenvector Spectral (KL)
Communication: 5 / 3 Bandwidth: 9 / 28
Start-Up Cost: 10 IBV: 10



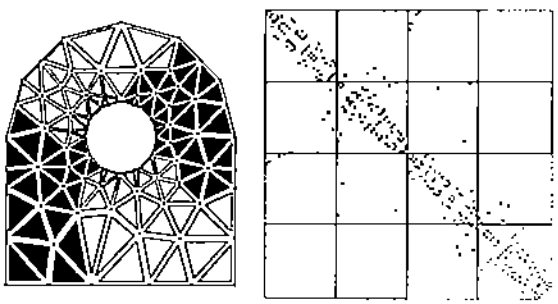
Cartesian - local optimum (KL)
Communication: 5 / 3 Bandwidth: 8 / 28
Start-Up Cost: 10 IBV: 10



Polar - recursive bisection (KL)
Communication: 14 / 9 Bandwidth: 6 / 26
Start-Up Cost: 42 IBV: 25



Polar - local optimum (KL)
Communication: 6 / 4 Bandwidth: 6 / 24
Start-Up Cost: 12 IBV: 11



Inertia - first eigenvector (KL)
Communication: 10 / 5 Bandwidth: 8 / 25
Start-Up Cost: 30 IBV: 15

Figure 8.9: The performance and sparse matrix of the considered mesh splitting algorithms with Kernighan-Lin refinement for the engine rod head (117m, 4p).

A-4 Algorithms of initial constructive domain decomposition

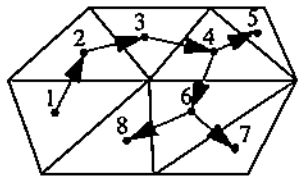
Pseudo-Code for the so called *constructive algorithm* that constructs the partitioning of the domain on a given finite element mesh graph G = (V, E) are listed. Let N_n and N_s describe the number of nodes in the graph G, and number of subdomains respectively.

A-4.1 Depth - First Search algorithm [Baase 88]

Depth - first search, which can be simply described by a recursive algorithm, is a generalization of pre-order traversal of trees. When an element is first visited and becomes part of the depth - first tree, it recursively searches its children, if such exist. Then the traversal scheme backs up to it and branches out in a different direction several more times.

Depth_First_Search(N_{start}):

Visit and mark N_{start} with partitioning number;
While there is an unmarked node A adjacent to N_{start} Do {
    Depth_First_Search(A);
}

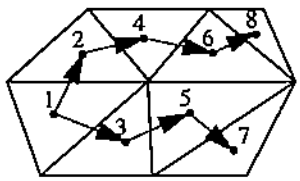


A-4.2 Breadth - First Search algorithm [Baase 88]

In a Breadth - first search, meshes are visited in the order of increasing distance from the starting point, where distance is simply the number of adjacency edges in a shortest path.

Breadth_First_Search(N_{start}):

Initialize queue Q to be empty;
Visit and mark N_{start} with partitioning number;
Insert N_{start} into Q;
While Q is non-empty Do {
    A = Remove_From_Queue(Q);
    For each unmarked node B adjacent to A Do {
        Visit and mark B with partitioning number;
        Insert B into Q;
    }
}



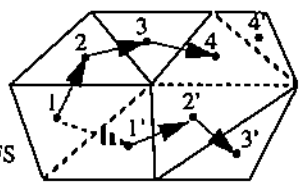
A-4.3 Domain - Wise Neighborhood Search algorithm [Farh 88]

In strip-wise decomposition, the direction in which the sorting of *edge distance* performed is fixed through the splitting procedure. While in domain-wise decomposition, the splitting direction is switched according to the relative searching scheme.

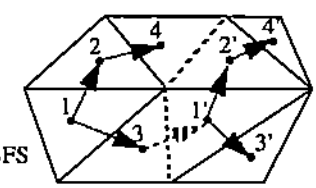
Domainwise_Search(N_{start}):

$N_0 = N_{start}$; /* starting node */
$N_p = N_n / N_S$; /* number of nodes in each subdomain */
For i = 1 to N_s Do { /* P_i is the i^{th} partition */
    N = a boundary node of P_{i-1} or N_0 if first partition;
    P_i = DFS/BFS(N) with the first unmarked N_p nodes;
}

domain-wise DFS



domain-wise BFS



A-4.4 Pseudo-Peripheral Node Locating algorithm [Pissan 84], [Georg 79]

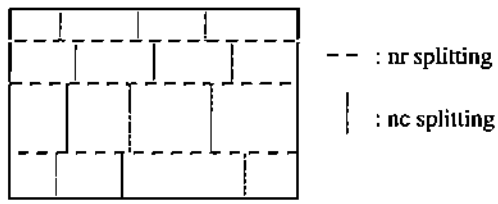
Following is the algorithm for finding a pseudo-peripheral node with a large eccentricity,

```
Pseudoperipheral_Node(G):

Initialize queue Q to be empty;
N = a node in G with the minimum degree;
Insert N into Q;
While Q is non-empty Do {
    A = Remove_From_Queue(Q);
    DFS/BFS(A);
    For each node B (in the order of increasing degree) at the deepest level of the current tree Do {
        DFS/BFS(B);
        if either one or both of   1. tree(B) has narrower tree level,
                                   2. tree(B) has longer tree depth. Do {
            Insert B into Q;
        }
    }
}
Return root of the current selected tree;
```

A-4.5 Fixed Non-Recursive Bisection of Cartesian Axis Splitting

For the non-recursive splitting scheme, the direction in which the sorting of coordinates is performed is fixed through each dimensional splitting procedure.

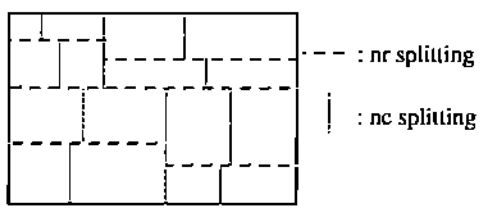
```
Fixed_RowColumn_Cartesian():

nr = N_r;  nc = N_c;        /* predefined row, column number */
Sort their Y coordinates;
Split the domain into nr subdomains along the Y axis;
For each splitted subdomain Do {
    Sort their X coordinates;
    Split the domain into nc subdomains along the X axis;
}
```



A-4.6 Recursive Bisection of Cartesian Axis Splitting

For this version, the recursive bisection is carried out until the number of subdomains reaches a predefined number.

```
Recursive_RowColumn_Cartesian():

nr = N_r;  nc = N_c;          /* predefined row, column number */
While nr or nc is larger than 1 Do {
    If ((nr = nr / 2) > 0) {
        For each splitted subdomains Do {
            Sort their Y coordinates;
            Split the domain into 2 subdomains along the Y axis;
        }
    }
    If ((nc = nc / 2) > 0) {
        For each splitted subdomain Do {
            Sort their X coordinates;
            Split the domain into 2 subdomains along the X axis;
        }
    }
}
```

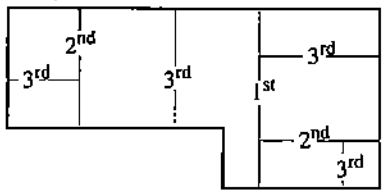


A-4.7 Longest Expansive Bisection of Cartesian Axis Splitting [Simon 91]

The algorithm chooses the bisecting direction by determining the longest expansion of domain to preserve a good aspect ratio of subdomains.

Longest_Cartesian():

```
ns = Ns;                        /* number of subdomains */
While ns larger than 1 Do {
    For each splitted subdomain Do {
        D = the longest expansion of X/Y/Z;
        Sort their D coordinates;
        Split the domain into 2 subdomains along the D axis;
    }
    ns /= 2;
}
```



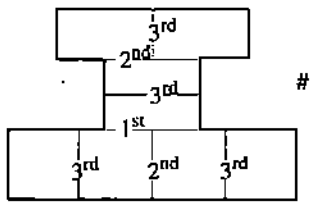
: split level number

A-4.8 Optimal Bisection of Cartesian Axis Splitting [Wu 93-1]

The splitting scheme selects the bisect direction by comparing the 'communication cost' produced by all possible dimensions. Then we choose the one that causes the least cost between these two new generated subdomains.

Optimal_Cartesian():

```
ns = Ns;                        /* number of subdomains */
While ns larger than 1 Do {
    For each splitted subdomain Do {
        Sort their X coordinates;
        Split the domain into 2 subdomains along the X axis;
        Compute the communication bisection width BWx;
        Sort their Y coordinates;
        Split the domain into 2 subdomains along the Y axis;
        Compute the communication bisection width BWy;
        Select the one that has smaller bisection width;
    }
    ns /= 2;
}
```



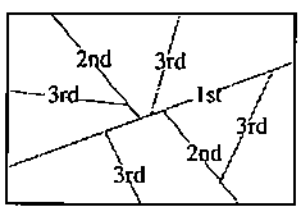
: split level number

A-4.9 Inertia Axis Splitting [Lori 88], [Willia 92]

This scheme first pre-computes the main symmetry axis according to the coordinate of nodes, then, it splits the domain into several subdomains along this axis. The following algorithm shows the splitting scheme of the inertia axis decomposition:

Inertia_Axis():

```
ns = Ns;                        /* number of subdomains */
nd = Nd;                        /* number of divide */
While ns is larger than 1 Do {
    For each splitted subdomain Do {
        Compute the main symmetry axis α;
        Split the domain into nd subdomains along axis α;
    }
    ns /= nd;
}
```



: split level number

The computation of the main symmetry axis α is as follows,

Computation of the main symmetry axis:

Let A be the (N_n x 3) matrix of the mesh coordinates that belong to the current domain with the original point as either:

1. Centre of Inertia of current meshes.
2. Centre of Mass of current meshes.
3. User specified.

The main symmetry axis is given by the Eigenvector corresponding to the largest Eigenvalue of A^TA.

A-5 Procedures for subdomain boundary linking

Two routines are needed for connecting the new boundary for each subdomain before proceeding with the final mesh generation in parallel. The related pseudo-code is as follows:

A-5.1 Boundary Linking algorithm [Wu 93-1]

The procedure locates all the polygon lists, including outer boundary polygon and hole polygon. Where E is the element mesh after the domain decomposition.

Link_Boundary(E):

```
Unmark all element mesh E;
While there is an unmarked element mesh E that is on the boundary Do {
    P = new polygon;
    Mark E and insert it into vertices list of P;
    E1 = E;
    Do {
        Search an unmarked element mesh E2 that is on the boundary and adjacent to E1;
        Mark E2 and insert it into vertices list of P;
        E1 = E2;
    } While (E1 != E);
}
```

A-5.2 Polygon-Hole Locating algorithm [Wu 93-1]

The routine identifies the outer boundary polygon and the hole polygon by the polygon locating recognition algorithm. Let P be the polygon list located by the previous procedure. We have,

Boundary_Hole(P):

```
For each polygon P1 ∈ P Do {
    For each polygon P2 ∈ P except P1 Do {
        If (Locate_Poly(P1, P2) == "Inside") {
            Link P1 to the hole list of P2;
            Break;
        }
    }
}
```

A-7 Algorithms of final refined domain decomposition

Pseudo-Code of the four widely used strategies for the partitioning refinement problem are the *Kernighan-Lin (KL) algorithm* [Krish 84], [Kern 70], the *Simulated Annealing (SA) algorithm* [John 89], [Kirk 83], the *Stochastic Evolution (SE) algorithm* [Saab 91], and the *Tabu Search (TS) algorithm* [Hertz 87], [Glov 85].

A-7.1 Kernighan - Lin Heuristic algorithm [Kern 70]

The K-L scheme is trying to locate the best k exchange pairs of nodes by constructing a sequence of gains. Let $C_{ij} = 1$ if nodes i and j are adjacent, 0 otherwise. We have,

Kernighan-Lin(subdomain(A), subdomain(B)):

Repeat {

Compute $D_a = \sum_{x \in B} C_{ax} - \sum_{y \in A} C_{ay}$ & $D_b = \sum_{x \in A} C_{bx} - \sum_{y \in B} C_{by}$ for each a ∈ A and b ∈ B;

For i = 1 to minimum(|A|, |B|) Do {

Select $a_i \in A$ and $b_j \in B$ such that gain $g_i = D_{ai} + D_{bi} - 2C_{ai,bi}$ is maximum;

Lock a_i & b_i;

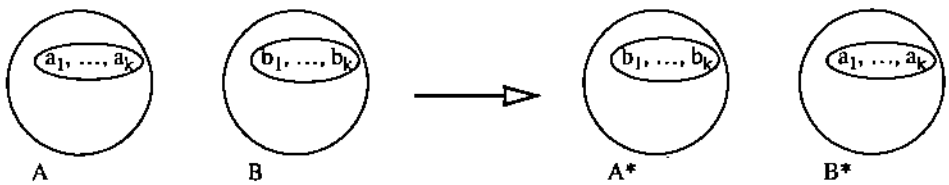
Recompute D values of Unlocked part of A & B, D_a += $2C_{a,ai} - 2C_{a,bi}$, D_b += $2C_{b,bi} - 2C_{b,ai}$;

}

Choose k to maximize $G = \sum_{i=1}^{k} g_i$;

Move $a_1, \ldots, a_k$ to B & $b_1, \ldots, b_k$ to A;

} Until G ≤ 0;



A-7.2 Multi-domain Kernighan - Lin Heuristic algorithm [Kern 70]

To optimize the multiple subdomains, we need the multi-domain version as follows:

Multi_Kernighan-Lin(N_s of subdomains):

Initial all elements of triangle matrix KL[N_s][N_s] to 1;
While there is an element KL[i][j] equals to 1 Do {
Kernighan_Lin(subdomain(i), subdomain(j));
If any exchange in the K-L procedure {
Set all KL[i][k] & KL[k][j] to 1, where k = 1, ..., N_s;
} Else {
Set KL[i][j] to 0;
}
}

A-7.3 Simulated Annealing algorithm [Kirk 83]

The SA scheme is an approach that attempts to prevent a poor local optimum by allowing an occasional uphill move. Let P be the initial partition, and its neighbor P_{new} is generated from P by a single move between subdomains. The cost of the partition is simply the number of edges.

Simulated_Annealing(P):

```
T = T0;                                  /* initial temperature */
K = K0;                                  /* initial logarithmic schedule step > 1 */
Repeat {
    Pnew = Random neighbor of P;
    gain = cost(P) - cost(Pnew);
    If (gain ≥ 0) or (random(0, 1) < e^(gain/T)) {
        P = Pnew;
    }
    T = either   α T;                    /* geometric cooling schedule, with constant α */
        or       (log (K) / log (K + 1)) T ;   /* logarithmic cooling schedule */
    K++;
} Until predefined criterion satisfied;
Return P;
```

A-7.4 Revised version of Simulated Annealing algorithm [Saab91], [John 89]

A double loop revised version follows that usually converges fast compared to the original one.

Revised_SA(P):

```
T = T0;                                  /* initial temperature */
K = K0;                                  /* initial logarithmic schedule step > 1 */
Repeat {
    Repeat {
        Pnew = Random neighbor of P;
        gain = cost(P) - cost(Pnew);
        If (gain ≥ 0) or (random(0, 1) < e^(gain/T)) {
            P = Pnew;
        }
    } Until predefined criterion satisfied;
    T = either   α T;                    /* geometric cooling schedule, with constant α */
        or       (log (K) / log (K + 1)) T ;   /* logarithmic cooling schedule */
    K++;
} Until T < Tf;                          /* frozen */
Return P;
```

A-7.5 Stochastic Evolution algorithm [Saab 91].

The SE scheme is an *adaptive heuristic* scheme that allows the uphill move probability to be updated whenever it is necessary.

```
Stochastic_Evolution(P):

P_best = P;
u = u_0;                          /* uphill move allowance, e.g. u_0 = 2. */
R = R_0;                          /* stopping criterion steps. */
For ρ = 1 to R Do {
    For each possible neighbor P_new of PDo {
        gain = cost(P) - cost(P_new);
        If (gain > random(-u, 0)) {
            P = P_new;
        }
    }
    If "net gain" ≤ 0 {
        Increase u;               /* increase uphill probability, e.g. u++ */
    } Else {
        u = u_0;
    }
    If cost(P) < cost(P_best) {
        P_best = P;
        ρ -= R;                   /* good improvement, do R more times. */
    }
}
Return P_best;
```

A-7.6 Tabu Search algorithm [Hertz 87], [Golv 85]

The unique feature of the TS scheme is the construction of a Tabu list. It is a list of Tabu moves that prevent the searching procedure from being cyclically stuck.

```
Tabu_Search(P):

T = Initial Tabu list;            /* e.g. |T| = 7 */
R = R_0;                          /* stopping criterion steps. */
For i = 1 to R Do {
    Repeat {
        P_new = the best neighbor of P with move P->P_new ∉ T or cost(P_new) < cost(P);
    } Until cost(P_new) < cost(P) or no more possible neighbor;
    P = P_new;
    Update T by inserting P->P_new & removing the oldest element;
}
Return P;
```

A-7.7 Parallel Mob Heuristic algorithm [Sava 91]

The Mob scheme has many of the features of the previously discussed algorithms but is better for exploiting available parallelism. Let N_n be the total number of nodes in subdomains. We have,

Parallel_Mob(subdomain(A), subdomain(B)):

```
R = R0;                              /* stopping criterion steps. */
Ms = Initial sequence of Mob schedule;   /* |Ms| = R, and all elements are in [1, Nn/2]. */
                                     /* Normally, in decreasing order. */
s = 1;
For i = 1 to R Do {
    Randomly select Ms[s] of 'good' nodes X & Y in parallel by random numbers r1(i) & r2(i);
    A = A - X + Y;
    B = B - Y + X;
    If gain < 0 {
        s++;
    }
}
```

References

[Wu 93-1] Poting Wu and Elias N. Houstis, "Parallel electronic prototyping of physical objects", *CAPO Technical Report, Purdue University, Department of Computer Sciences*, CER-**93-19**, (April 1993) 1-48.

[Wu 93-2] Poting Wu, Elias N. Houstis and John R. Rice, "Geometry as a basis for parallel analysis and design of physical objects", *Second U.S. National Congress on Computational Mechanics, USACM*, (August 1993) Abstracts-72.

[Chri 91] N. P. Chrisochoides, E. N. Houstis and C. E. Houstis, "Geometry based mapping strategies for PDE computations", *Proceedings of the International Conference on Supercomputing*, Cologne-Germany, (June 1991) 128-135.

[Lohn 92] Rainald Lohner, Jose Camberos and Marshal Merriam, "Parallel unstructured grid generation", *Computer Methods in Applied Mechanics and Eng*ineering **95**, (1992) 343-357.

[Khan 91] A. I. Khan and B. H. V. Topping, "Parallel adaptive mesh generation", *Computing Systems in Engineering* **2-1**, (1991) 75-101.

[Cheng 89] Fuhua Cheng, Jerzy W. Jaromczyk, Junnin-Ren Lin, Shyue-Shian Chang and Jei-Yeou Lu, "A parallel mesh generation algorithm based on the vertex label assignment scheme", *International Journal for Numerical Methods in Engineering* **28**, (1989) 1429-1448.

[Lo 91] S. H. Lo, "Automatic mesh generation and adaptation by using contours", *International Journal for Numerical Methods in Engineering* **31**, (1991) 689-707.

[Bykat 76] A. Bykat, "Automatic generation of triangular grid: I-Subdivision of a general polygon into convex subregions. II-Triangulation of convex polygons", *International Journal for Numerical Methods in Engineering* **10**, (1976) 1329-1342.

[Sadek 80] Edward A. Sadek, "A scheme for the automatic generation of triangular finite elements", *International Journal for Numerical Methods in Engineering* **15**, (1980) 1813-1822.

[Delj 90] K. Deljouie-Rakhshandeh, "An approach to the generation of triangular grids possessing few obtuse triangles", *International Journal for Numerical Methods in Engineering* **29**, (1990) 1299-1321.

[Samet 84] Hanan Samet, "The quadtree and related hierarchical data structures", *Computing Surveys* **16-2**, (1984) 187-260.

[Samet 82] Hanan Samet, "Neighbor finding techniques for images represented by quadtrees", *Computer Graphics and Image Processing* **18**, (1982) 37-57.

[Samet 85] Hanan Samet and Clifford A. Shaffer, "A model for the analysis of neighbor finding in pointer-based quadtrees", *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-7-6**, (1985) 717-720.

[Samet 89] Hanan Samet, "Neighbor finding in images represented by octrees", *Computer Vision, Graphics, and Image Processing* **46**, (1989) 367-386.

[Mitch 87] William F. Mitchell, "A comparison of adaptive refinement techniques for elliptic problems", *Technical Report, University of Illinois at Urbana-Champaign, Department of Computer Science*, UIUCDCS-R-**87-1375**, (September 1987) 1-14.

[Zienk 87] O. C. Zienkiewicz and J. Z. Zhu, "A simple error estimator and adaptive procedure for practical engineering analysis", *International Journal for Numerical Methods in Engineering* **24**, (1987) 337-357.

[Garey 76] M. R. Garey, D. S. Johnson and L. Stockmeyer, "Some simplified NP-complete graph problems", *Theoretical Computer Science* **1**, (1976) 237-267.

[Farh 93-1] Charbel Farhat and Michel Lesoinne, "Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics", *International Journal for Numerical Methods in Engineering* **36**, (1993) 745-764.

[Venk 92] V. Venkatakrishnan and Horst D. Simon, "A MIMD implementation of a parallel Euler solver for unstructured grids", *The Journal of Supercomputing* **6**, (1992) 117-137.

[Farh 93-2] Charbel Farhat and Horst D. Simon, "TOP/DOMDEC - a software tool for mesh partitioning and parallel processing", *Technical Report, NASA Ames Research Center*, RNR-**93-011**, (June 1993) 1-28.

[Simon 91] H. D. Simon, "Partitioning of unstructured problems for parallel processing", *Computing Systems in Engineering* **2-2/3**, (1991) 135-148.

[Malon 88] James G. Malone, "Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessor computers", *Computer Methods in Applied Mechanics and Engineering* **70**, (1988) 27-58.

[Bacch 91] L. Bacchelli Montefusco and C. Guerrini, "A domain decomposition method for scattered data approximation on a distributed memory multiprocessor", *Parallel Computing* **17**, (1991) 253-263.

[Lori 88] M. Loriot and L. Fezoui, *Mesh-splitting preprocessor*, unpublished manuscripts, (1988).

[Willia 92] Roy Williams, "Parallel meshes for complex geometry", *PML*, (February 12, 1992) 302-312.

[Baase 88] Sara Baase, "Graphs and Digraphs", *Computer Algorithms: Introduction to Design and Analysis*, Reading, MA: Addison-Wesley, (1988) Ch.**4**, 145-207.

[Pissan 84] Sergio Pissanetsky, "Ordering for Gauss elimination: Symmetry matrices", *Sparse Matrix Technology*, Orlando: Academic Press, (1984) Ch.**4**, 94-158.

[Georg 79] Alan George and Joseph W. H. Liu, "An implementation of a pseudoperipheral node finder", *ACM Transactions on Mathematical Software* **5-3**, (1979) 284-295.

[Gibbs 76] Norman E. Gibbs, William G. Poole, JR. and Paul K. Stockmeyer, "An algorithm for reducing the bandwidth and profile of a sparse matrix", *SIAM J. Numer. Anal* **13-2**, (1976) 236-250.

[Cuth 69] E. Cuthill and J. McKee, "Reduce the bandwidth of sparse symmetric matrices", *Proceedings of 24th National Conference, ACM* **P-69**, (1969) 157-172.

[Chan 80] W. M. Chan and Alan George, "A linear time implementation of the reverse Cuthill-McKee algorithm", *BIT* **20**, (1980) 8-14.

[Georg 78] Alan George and Joseph W. H. Liu, "Algorithms for matrix partitioning and the numerical solution of finite element systems", *SIAM J. Numer. Anal.* **15-2**, (1978) 297-327.

[Akyuz 68] Fevzican A. Akyuz and Senol Utku, "An automatic node-relabeling scheme for bandwidth minimization of stiffness matrices", *AIAA Journal* **6-4**, (1968) 728-730.

[Rosen 68] Richard Rosen, "Matrix bandwidth minimization", *Proceedings of 23rd National Conference, ACM* **P-68**, (1968) 585-595.

[Alway 65] G. G. Alway and D. W. Martin, "An algorithm for reducing the bandwidth of a matrix of symmetrical configuration", *The Computer Journal* **8-3**, (1965) 264-272.

[Jenn 66] Alan Jennings, "A compact storage scheme for the solution of symmetric linear simultaneous equations", *The Computer Journal* **9-3**, (1966) 281-285.

[Farh 88] Charbel Farhat, "A simple and efficient automatic FEM domain decomposer", *Computers & Structures* **28-5**, (1988) 579-602.

[AlNas 91] M. Al-Nasra and D. T. Nguyen, "An algorithm for domain decomposition in finite element analysis", *Computers & Structures* **39-3/4**, (1991) 277-289.

[Fied 73] Miroslav Fiedler, "Algebraic connectivity of graphs", *Czechoslovak Mathematical Journal* **23-98**, (1973) 298-305.

[Fied 75-1] Miroslav Fiedler, "A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory", *Czechoslovak Mathematical Journal* **25-100**, (1975) 619-633.

[Fied 75-2] Miroslav Fiedler, "Eigenvectors of acyclic matrices", *Czechoslovak Mathematical Journal* **25-100**, (1975) 607-618.

[Barn 93] Stephen T. Barnard and Horst D. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems", *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, (March 1993) 711-718.

[Hend 93] Bruce Hendrickson and Robert Leland, "An improved spectral load balancing method", *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, (March 1993) 953-961.

[Leete 93] Charles A. Leete, Barry W. Peyton and Richard F. Sincovec, "Toward a parallel recursive spectral bisection mapping tool", *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, (March 1993) 923-928.

[Mori 87] R. Morison and S. Otto, "The scattered decomposition for finite elements", *Journal of Scientific Computing* **2-1**, (1987) 59-76.

[Leigh 92] F. Thomson Leighton, "Hypercubes and related networks", *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*, San Mateo, CA: Morgan Kaufmann Publishers, (1992) Ch.**3**, 389-785.

[Kern 70] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs", *The Bell System Technical Journal* **49**, (February 1970) 291-307.

[Krish 84] Balakrishnan Krishnamurthy, "An improved min-cut algorithm for partitioning VLSI networks", *IEEE Transactions on Computers* **C-33-5**, (May 1984) 438-446.

[Vand 93] D. Vanderstraeten, O. Zone, R. Keunings and L. A. Wolsey, "Non-deterministic heuristics for automatic domain decomposition in direct parallel finite element calculations", *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, (March 1993) 929-932.

[John 89] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon, "Optimization by simulated annealing: An experimental evaluation; Part I, Graph partitioning", *Operations Research* **37-6**, (November-December 1989) 865-892.

[Kirk 83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing", *Science* **220-4598**, (13 May 1983) 671-680.

[Saab 91] Youssef G. Saab and Vasant B. Rao, "Combinatorial optimization by stochastic evolution", *IEEE Transactions on Computer-Aided Design* **10-4**, (1991) 525-535.

[Hertz 87] A. Hertz and D. de Werra, "Using Tabu search techniques for graph coloring", *Computing* **39**, (1987) 345-351.

[Glov 85] F. Glover and C. McMillan, "Interactive decision software and computer graphics for architectural and space planning", *Annals of Operations Research* **5**, (1985/6) 557-573.

[Byun 93] Hyeran Byun, "Neurocomputing on distributed memory machines", *Purdue University, Department of Computer Sciences, Ph.D thesis*, to appear.

[Sava 91] John E. Savage and Markus G. Wloka, "Parallelism in graph-partitioning", *Journal of Parallel and Distributed Computing* **13**, (1991) 257-272.

[Willia 91] Roy D. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations", *Concurrency: Practice and Experience* **3-5**, (1991) 457-481.

[Walsh 93] Chris Walshaw and Martin Berzins, "Enhanced dynamic load-balancing of adaptive unstructured meshes", *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, (March 1993) 971-978.

[Curra 92] Mark C. Curran and Myron B. Allen, "Domain-decomposition approach to local grid refinement in finite element collocation", *Numerical Methods for Partial Differential Equations* **8**, (1992) 341-355.

[Geist 92] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "A users' guide to PICL, a portable instrumented communication library", *Technical Report, Oak Ridge National Laboratory*, Oak Ridge, TN, ORNL/TM-**11616**, (February 1992) 1-22.

[Geist 90] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "PICL: a portable instrumented communication library, C reference manual", *Technical Report, Oak Ridge National Laboratory*, Oak Ridge, TN, ORNL/TM-**11130**, (July 1990) 1-146.

[Heat 93-1] Michael T. Heath and Jennifer Etheridge Finger, "ParaGraph: A tool for visualizing performance of parallel programs", *Oak Ridge National Laboratory*, Oak Ridge, TN, (August 1993) 1-50.

[Heat 93-2] Michael T. Heath, "Recent developments and case studies in performance visualization using ParaGraph", In G. Haring and G. Kotsis (Eds.), *Performance Measurement and Visualization of Parallel Systems*, Amsterdam, The Netherlands: Elsevier Science Publishers, (1993) 175-200.

[Heat 91] Michael T. Heath and Jennifer Etheridge Finger, "Visualizing the performance of parallel programs", *IEEE Software* **8-5**, (1991) 29-39.